Evaluación y diseño de aplicaciones multimedia con tecnología ICE y Android.

UNIVERSIDAD CARLOS III

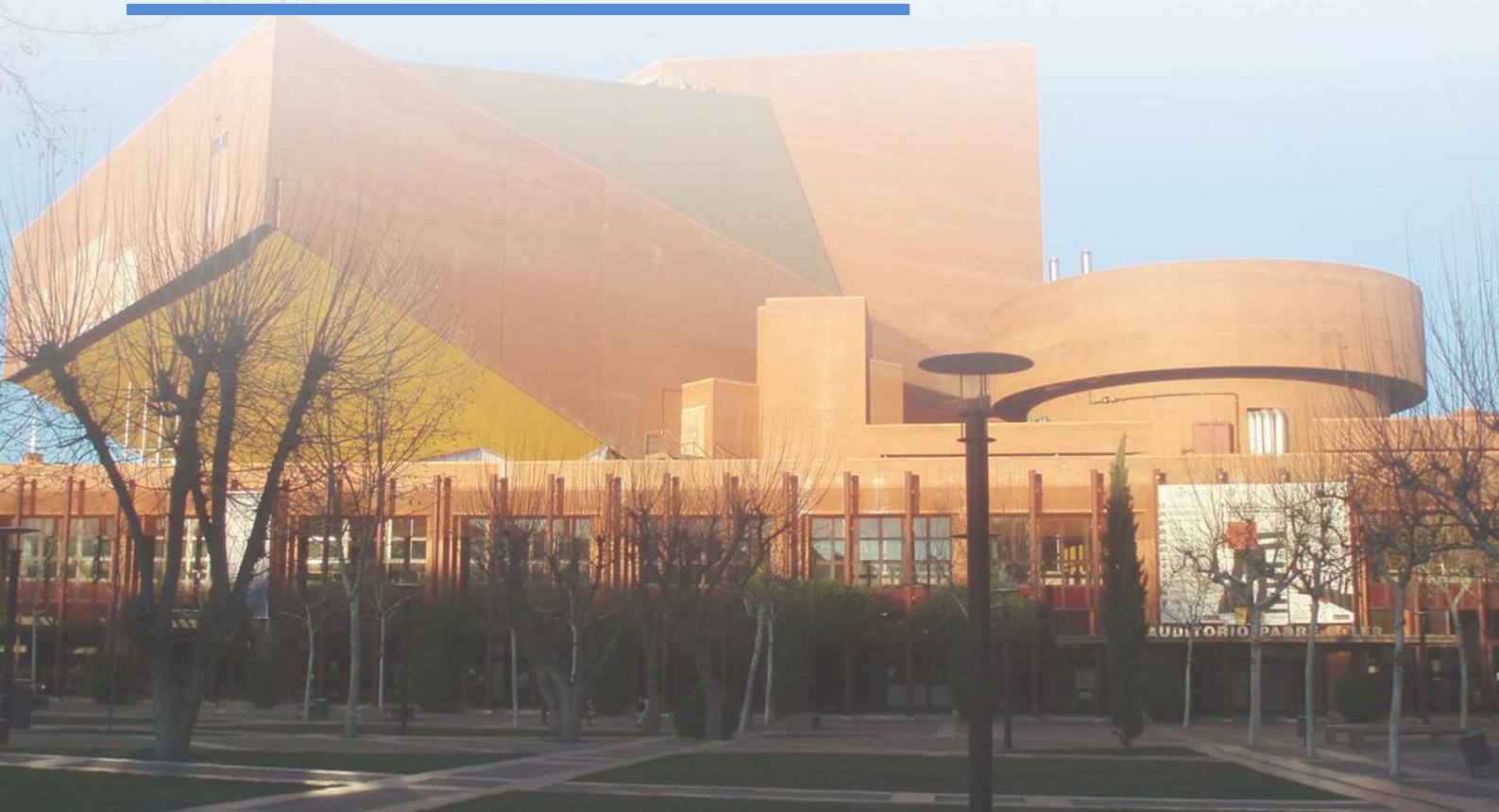
Escuela Politécnica

Ingeniería Técnica de Telecomunicación

TELEMÁTICA

Autor: Israel Cendrero Sánchez

Tutora: Iria Estévez Ayres



Título: Evaluación y diseño de aplicaciones multimedia con tecnología ICE y Android.

Autor: Israel Cendrero Sánchez

Tutor: Iria Estévez Ayres

EL TRIBUNAL

Presidente: María del Carmen Fernández Panadero

Secretario: Ricardo Romeral Ortega

Vocal: Marcelino Lázaro Teja

Realizado el acto de defensa del Proyecto Fin de Carrera el día **17 de Febrero de 2011** en Leganés, en la Escuela Politécnica Superior de la **Universidad Carlos III de Madrid**, acuerda otorgarle la **CALIFICACIÓN** de:

Fdo: Presidente Fdo: Vocal Fdo: Secretario

Aunque sé que ya nunca podrás leer esto,
va por ti Rodri.

Agradecimientos

En primer lugar (el orden da igual, por algún sitio hay que empezar) es darle las gracias a mi tutora Iria por todo el tiempo dedicado ya que conmigo no va a ver un duro. Ha sido un honor y un placer trabajar contigo.

A mis padres Jose y Merce, y a mi hermana Jara, que me han estado aguantando en casa (y me siguen aguantando, aunque les quede poco) y en general a toda mi familia (que me habrán aguantado en algún momento).

Y a mis amigos de toda la vida Luismi, Paola, Cristian, Lorena, Vivas, Elsa, Cris, Laila, Bufi… seguro que me dejo a alguno, pero vaya, ya me lo echaréis en cara.

A mis primix de Vázquez de Mella, que me acogieron en momentos difíciles y me dieron lo que más necesitaba en ese momento: Fiesta y evasión. A la mejor amiga que he podido tener nunca: Berta, y a Celia, Cristi, Pinche, Txus y Jorge. Y en especial a Fido, por darme cobijo en Londres y por algo mejor: presentarme a toda la gente mencionada en este párrafo.

Hablando de Londres, no puedo pasar por aquí sin darle las gracias por todos los favores, cobijos y momentos prestados por tierras británicas y tricantinas. Muchas gracias por ser tú misma y ser así, Ali.

Y ahora tengo mentar a mis compañeros de clase Javi, Jesú, Sergio, Carlos, Paola, Ananas, Marce, Gema, Guille, Juanal, Marta, Bowen, Yawe, Jorge, Sergio, Raquel, Alvarito, Ana, Samu, Bowen y Marcos por haberme aguantado día a día y por toda la ayuda prestada. Es incalculable

Pero de la universidad, en especial a Raúl, Jan, San, Pepelu, Patri, Mamen y Lara, que me habéis dado el empujón final para terminar las asignaturas. Si no hubiera sido por vuestra ayuda y vuestros ánimos, todavía estaba sacándome TDI o Tendencias o seguiría mirando a ver cómo se instala el Eclipse para empezar el proyecto.

Tengo que mentar, por supuesto, a la gente que me está aguantando semana a semana (o no tanto, según nos dé, pero nuestras fiestas nos hemos pegado) de Valdemoro, gracias a Mari “Cookie”, Sergy “Hazte un suplex”, María “Ramirez”, Bea “la de Felipe” y Ali “Aleces”.

Pero todos los agradecimientos del párrafo anterior y de esta hoja no tendrían ningún sentido si no agradeciera a Shei por estar ahí, por aparecer en mi vida y arreglarla del todo, por todos los momentos vividos, todos los viajes hechos y que nos quedan por hacer, por hacer del Sierra un icono más que una tartana, por enriquecerme musicalmente hasta cotas que jamás hubiera podido imaginar, por darme motivos para seguir dibujando y tocando la guitarra y por darme el mejor motivo para terminar la carrera, que es el empezar a pensar en independizarme contigo. Nos quedan muchos festivales a los que ir y muchos quebraderos de cabeza que superar juntos, pero de eso ahora no nos preocupamos, porque ahora…

Ahora empieza lo bueno.

Resumen

La compartición de fotos a través de las redes sociales es algo que está muy presente en el día a día de la gente que está acostumbrada a navegar en internet. No es tanto la privacidad de la que se disfruta al subir fotos a dichas redes sociales, de ahí nace la idea de ***SharePhoot****, una galería remota en la que se puede manejar , borrar o añadir fotos al antojo del usuario desde un terminal Android hasta el ordenador personal sin pasar por intermediarios.*

En este proyecto se estudia la tecnología middleware de comunicaciones ***ICE*** *la cual hace posible la interconexión de sistemas heterogéneos. Se estudia en detalle el protocolo que implementa esta tecnología y se lleva a cabo el desarrollo de una aplicación para probar todo lo estudiado.*

Se estudia también el sistema operativo de Google, Android, y se pone en práctica el uso de la tecnología ***ICE*** *sobre esta plataforma, desarrollando así, la aplicación* ***SharePhoot****, a la cual se le realizaron pruebas de rendimiento.*

Palabras clave: Android, SharePhoot, Red Social, ICE, middleware

Abstract

The photo sharing via social networks it's something quite present day by day in the non-rookies surfers of the internet. But there's no such privacy as they thought when they're uploading in these social networks. This is how ***SharePhoot****'s idea comes to a term. With SharePhoot, the users can upload, delete or handle pictures directly into their personal computers without intermediates.*

In this project, we study the middleware technology for communications ICE which makes possible the interconnection of heterogenic systems. We study in details the protocol implemented by this technology and then develop an application for proving all the lessons learnt.

We study also the Google's operating system, Android, and we use the ICE technology with this platform. Developing in that way, SharePhoot was used for making performance tests.

Keywords: Android, SharePhoot, Social network, ICE, middleware

Índice General

Capítulo 3: Estudio del protocolo ICE

Capítulo 4: Desarrollo de una aplicación basada en la tecnología ICE y android

Capítulo 5: Evaluación

Índice de figuras

Índice de tablas

Lista de acrónimos

ADT *Android Development Tools*

AIFF *Audio Interchange File Format*

API *Application Programming Interface*

ATW *Abstract Window Toolkit*

AU *Audio*

AVD *Android Virtual Device*

AVI *Audio Video Interleave*

BSD *Berkeley Software Distribution*

CORBA *Common Object Request Broker Architecture*

CPU *Computer Processor Unit*

DDMS *Dalvik Debug Monitor*

FPS *Frames Per Second*

FTP *File Transfer Protocol*

GPS *Global Positioning System*

GSM *Global System for Mobile communications*

GUI *graphical user interface*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

ICE *Internet Connection Engine*

ID *Identity*

IDE *Integrated Development Environment*

IDL *Interface description language*

IP *Internet Protocol*

JDK *Java Development Kit*

JMF *Java media Framework*

JPEG *Joint Photographic Experts Group*

MIDI *Musical Instrument Digital Interface*

MPEG *Moving Picture Experts Group*

OHA *Open Handset Alliance*

OS *Operating System*

PC *Personal Computer*

QT *QuickTime*

RFC *Request For Comments*

RIM *Research In Motion*

RMF *Rich Music Format*

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

RTP *Real Time Protocol*

RTSP *Real Time Streaming Protocol*

SD *Secure Digital*

SDK *Software Development Kit*

SLICE *Specification Language for ICE*

SMS *Short Message Service*

SOAP *Simple Object Access Protocol*

SP1 *Service Pack 1*

SSL *Secure Sockets Layer*

TCP *Transmission Control Protocol*

WAV *Waveform Audio File Format*

WCF *Windows Communication Foundation*

XML *Extensible Markup Language*

Capítulo 1

Introducción y motivación

1.1 Introducción

Este proyecto tiene como finalidad estudiar la tecnología **ICE** y sus posibles aplicaciones en el mundo de los teléfonos inteligentes, en concreto en aquéllos que tengan Android como sistema operativo.

Para ello, se realizará un estudio de la tecnología de comunicaciones **ICE**, desarrollado por la empresa de software libre **ZeroC**, y se estudiará en detalle el protocolo que implementa dicha tecnología, desarrollando una aplicación específica para su estudio.

Además, se realizó un estudio del sistema operativo de Google, Android, el cual se usó como plataforma para desarrollar una aplicación que funcionara sobre la plataforma **ICE** for Android.

1.2 Motivación

El aumento de cámaras de fotos en los móviles ha ido creciendo exponencialmente con el paso del tiempo y ya es una funcionalidad indispensable en cualquier terminal. Esta conclusión puede obtenerse simplemente mirando cualquier catálogo de móviles, y comprobar cuántos de ellos tienen cámara de fotos y cuantos no. Los teléfonos que no poseen una cámara, son considerados de gama baja, o de gama sencilla para los usuarios que utilicen el teléfono, no como un *gadchet* (1) si no como un teléfono sin más.

La pérdida o robo de un móvil en la actualidad, más allá de la pérdida de datos importantes, tanto como de contactos o mensajes de texto, es la pérdida de todas las imágenes que se han ido recopilando a lo largo de la vida del terminal.

Con esta motivación se ha desarrollado la aplicación **SharePhoot**, una aplicación que emplea el sistema operativo libre de Android, presentando el 5 de Noviembre de 2007 por Google junto con la fundación **OHA** (*Open Handset Alliance)*, un consorcio de 48 compañías de hardware, software y telecomunicaciones comprometidas con la promoción de estándares abiertos para dispositivos móviles.

Figura 1: Cartel promocional de Android

Consistente en una aplicación para la gestión de galerías de fotos, **SharePhoot** utiliza, aparte del citado sistema operativo de Google, la tecnología de *middleware* de comunicaciones **ICE**(*Internet Connection Engine*), desarrollado por **ZeroC** (2), y que permite la conectividad entre el ordenador personal del usuario y el terminal móvil.

Figura 2: Logo de ZeroC

1.3 Objetivos

Se establecen para el desarrollo de este proyecto los siguientes objetivos:

- Estudio del protocolo **ICE**
- Estudio de **ICE** en distintas plataformas
- Estudio de Android
- Estudio de **ICE** sobre Android
- Implementación de una aplicación que funcione en **ICE** sobre Android

1.4 Estructura del proyecto

En primera instancia, se pasará al análisis de la tecnología **ICE** tanto en su parte de protocolo, como en la parte de desarrollo, ya que este producto libre de **ZeroC** nos provee tanto de un **API** de desarrollo como de un protocolo completo el cual puede estudiarse como una capa eficiente y robusta de *middleware*. Así pues, en esta parte del proyecto, se realizarán diversas pruebas demostrando la eficacia de éste.

También se desarrollará una aplicación práctica para probar todo lo aprendido en el estudio de **ICE** para comprobar la utilidad del mismo.

Con **SharePhoot** se llega a la segunda parte, llevando a la práctica otra vez todo lo aprendido en cuanto a **ICE** pero mezclando dicha tecnología con el sistema operativo de Google, Android.

De tal forma que la estructura del proyecto queda:

- Capítulo 1: Introducción
- Capítulo 2: Estado del arte
- Capítulo 3: Estudio del protocolo **ICE**
- Capítulo 4: Desarrollo de una aplicación basada en la tecnología **ICE** y Android
- Capítulo 5: Evaluación
- Capítulo 6: Conclusiones y trabajos futuros
- Apéndices e información adicional

Capítulo 2

Estado del arte

2.1 Introducción

En este capítulo introducen los conceptos de las dos tecnologías que ocupan el grueso del proyecto realizado: la tecnología de *middleware* de comunicaciones **ICE** (3) y el sistema operativo para móviles, Android (4), a**c**tual competencia del *iPhone* de *Apple* (5) o la *BlackBerry* (6) de **RIM**(*Research In Motion*), la cual va ganando día a día más cuota de mercado.

Figura 3: Las tres grandes competidoras actuales

Dos tecnologías que aparentemente no tienen nada que ver, tales como una capa de *middleware* y un sistema operativo, combinadas pueden ser una gran herramienta para el desarrollo de aplicaciones libres.

Uno de los factores críticos a la hora de elegir estas tecnologías, ha sido la libre distribución de estas y la relativa novedad que suponen.

Por otra parte, que Android, está ganando día a día más defensores de su tecnología debido a la velocidad de ésta. Realmente, no deja de ser un sistema operativo para *smartphones* como el de *BlackBerry* o el del *iPhone*, pero a diferencia de sus competidores, éste vive bajo una licencia *Apache 2.0* (7) y el usuario puede programar sobre él usando todo su potencial de manera gratuita.

En lo referente a la tecnología **ICE**, se puede resumir en que es una tecnología de *middleware* libre que provee una serie de librerías y funcionalidades para permitir la interconexión de dos equipos, con las mismas premisas de **CORBA** (8)**,** mucho más ligera y sencilla de desarrollar que ésta, pero a la vez, menos potente.

2.2 Introducción a Android O.S(Google)

Android es de las últimas incorporaciones al mercado de los sistemas operativos para móviles, consiguiendo un éxito comparable, y a la vez mayor, que con el del iPhone de Apple. Aunque Android suele entenderse como un producto de Google, el desarrollo es promovido por la **OHA**, compuesta por empresas tan importantes como **HTC**, **Dell**, **Intel**, **Motorola**, **Sony Ericsson**, **Samsung**, **LG**, **T-Mobile**, **Nvidia** o **eBay** (9)**,** y es un proyecto de software libre, ya que gran parte de su código está bajo licencia **Apache 2.0**.

Una de las ventajas más claras de esta plataforma respecto a su competencia es la facilidad para desarrollar aplicaciones, existiendo versiones del **SDK**(*Software Development Kit*) para los tres sistemas operativos dominantes en el mercado de los ordenadores personales: **Windows, MacOS y GNU/Linux**. El hecho de que se pueda desarrollar usando cualquier terminal sin "pagar una cuota de desarrollo" ayuda a acercar a los programadores con poca experiencia a este sistema sin coste alguno. Si se quisiera profundizar en temas comerciales, tales como la publicación en el *AndroidMarket* (la tienda on-line de aplicaciones de Android) ya sí se pagaría una cuota.

Ofrece un gran rango de herramientas de desarrollo, un **API**(*Application Programming Interface*) potente, multitarea, versatilidad de las aplicaciones y la posible modificación del sistema operativo del terminal por uno personalizado. Alguna de las características adquiridas por su condición de software libre son las siguientes:

- **Multitud de versiones**: la gran variedad de dispositivos disponible con distintas capacidades, las múltiples versiones de Android y distintos niveles de **SDK,** puede complicar el desarrollo de las aplicaciones, si bien es cierto que hay formas de especificar las necesidades mínimas de una aplicación.

- **Personalización de operadoras**: las operadoras suelen modificar las aplicaciones, personalizando el teléfono y moldeándolo a las necesidades de la compañía en vez de orientarlas al usuario, llegando a extremos de imposibilitar al usuario la desinstalación de dichas aplicaciones independientemente de las necesidades del usuario (10)**.**

- **Obsolescencia dependiente**: Aunque el terminal pueda soportar factiblemente una versión nueva, si una empresa perteneciente a la **OHA** decide no dar soporte a nuevas versiones, quedaría desprotegido de posibles fallos.

A pesar de los inconvenientes de un proyecto de software libre, la comunidad de desarrollo de Android es muy amplia, y se nutre de su condición de software libre para crecer a grandes pasos, permitiendo, por ejemplo, tener funcionalidades de última generación en teléfonos Android antiguos que no han tenido obligatoriamente que actualizar su sistema operativo.

La libertad de uso y que, sin pertenecer a la **OHA**, muchos fabricantes puedan utilizar Android ha logrado que existan dispositivos en el mercado que se puedan adaptar a las necesidades de cada consumidor.

En la actualidad Android es la segunda plataforma con más cuota de mercado internacional y cuenta con la segunda mayor tienda de aplicaciones, tras *AppStore* de Apple (11), aunque, en previsiones de mercado y en crecimiento, gana ampliamente Android y se prevé que supere a su homóloga en poco tiempo como se puede ver en la Figura 4.

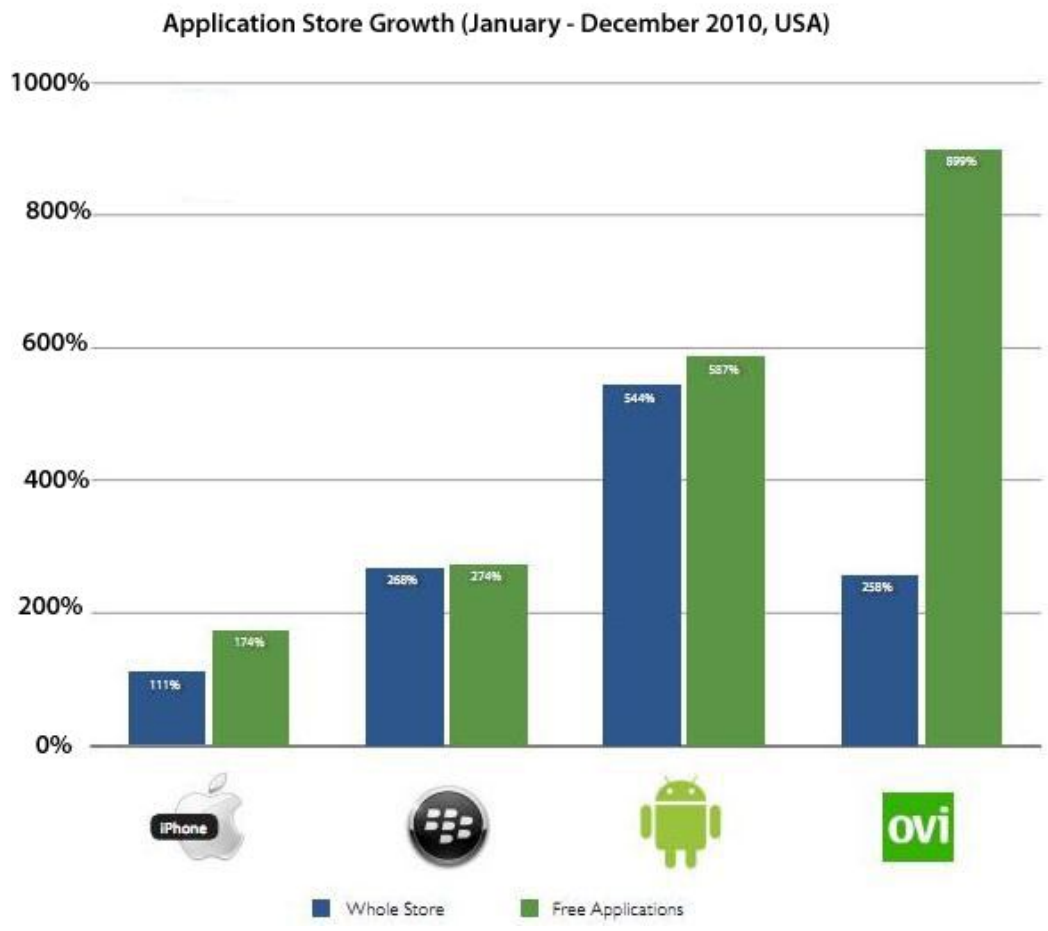


Figura 4: Crecimiento de las tiendas de aplicaciones a fecha diciembre 2010[]

2.2.1 Información de Android

2.2.1.1 Versiones

Como ya se ha explicado, Android posee una gran potencia en materia de desarrollo al ser libre y está en constante evolución. Prueba de ello es la gran comunidad que hay formada en torno a Android y a las modificaciones sobre las versiones oficiales de él. Entre muchos, las modificaciones publicadas en los foros de *CyanogenMods* (12).

Sobre la evolución de versiones en Android, podemos ver en el gráfico de la Figura 5cómo está la distribución de versiones actualmente (13)**.**

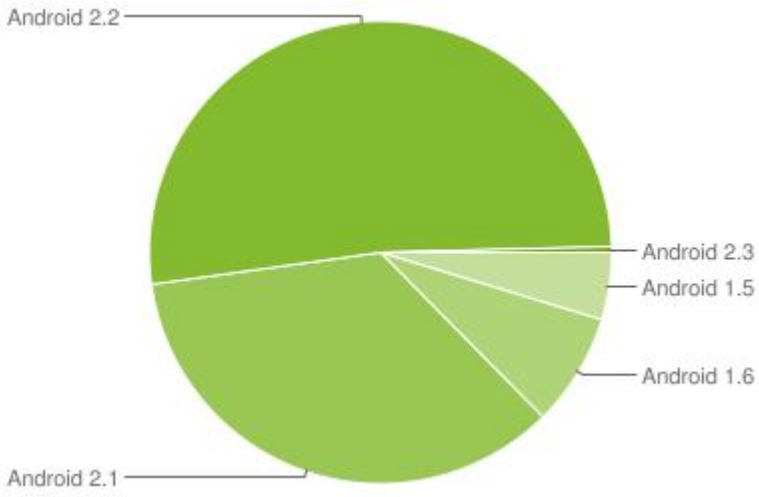


Figura 5: Distribución de versiones en Android

Como puede verse en el gráfico de la Figura 5 y su evolución en la gráfica de la figura Figura 6, la versión 2.2 (*Frozen Yogurth*) junto con la 2.1 (*Éclair*) es la más establecida. Esto ha sucedido en tan sólo 5 meses tras su salida en julio de 2010 (14) por lo que cabe esperar, que la distribución 2.3 (Gingerbread) lanzada en diciembre de 2010 (15), alcance cotas parecidas pasado un tiempo. Los datos en tanto por ciento se pueden consultar en la Tabla 1.

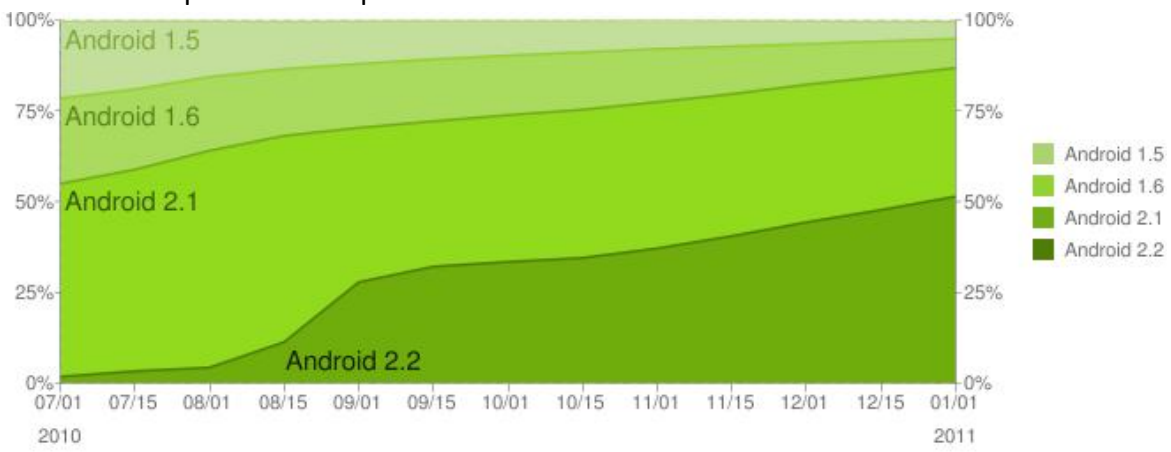


Figura 6: Evolución en el tiempo de las versiones de Android

Platform	API Level	Distribution
Android 1.5	3	4.7%
Android 1.6	4	7.9%
Android 2.1	7	35.2%
Android 2.2	8	51.8%
Android 2.3	9	0.4%

Tabla 1: Tabla de distribución de versiones de Android

2.2.1.2 Tamaños de pantalla y densidades

Una de las características que otorga a Android de una gran versatilidad, es la posibilidad de desarrollar sobre él independientemente del tamaño de la pantalla del terminal que se esté usando. Se puede ver la relación de tamaños de pantalla y densidades que ofrece el emulador de Android en la Tabla 2 y las equivalencias con las pulgadas en la Figura 7.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra high density (320), *xhdpi*
Small	QVGA (240x320)			
Normal	WQVGA400 (240x400) WQVGA432 (240x432)		WVGA800 (480x800) WVGA854 (480x854)	
Large		WVGA800* (480x800) WVGA854* (480x854)		
XLarge				

Tabla 2: Tipos de pantalla ofrecidos por el emulador de Android

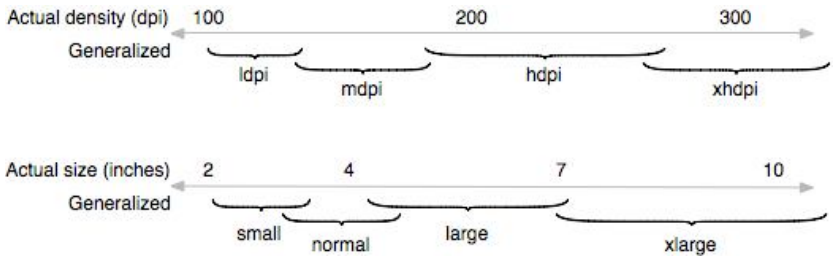


Figura 7: Descripción de las densidades de pantalla

En el gráfico de la Figura 8 y en la Tabla 3, se puede comprobar, que el rango de terminales empleados por los usuarios, varía entre una densidad de pantalla normal y alta ocupando más del **97%** del mercado (16), por lo que se considera, que es importante desarrollar en estas densidades dado que es lo más extendido.

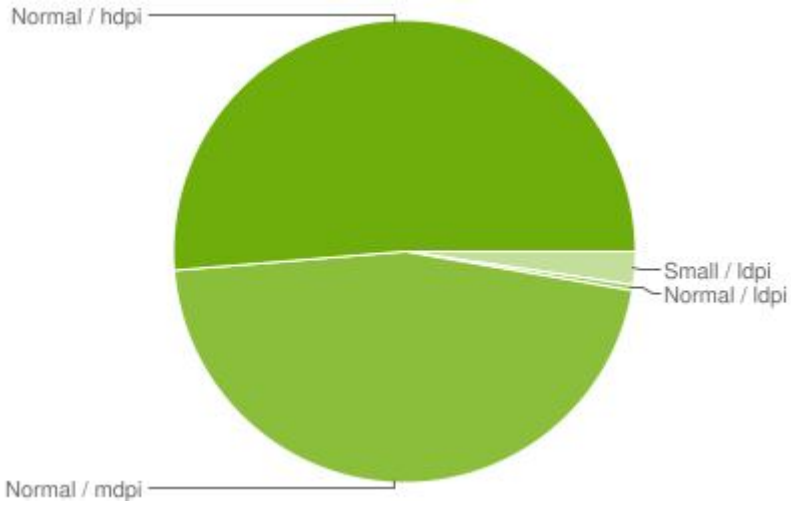


Figura 8: Gráfico de distribución de las características de las pantallas en Android

Screen Size / Density	Low Density	Medium Density	High Density
Small	2.3%		
Normal	0.4%	45.9%	51.2%
Large			

Tabla 3: Distribución de los tamaños/densidades de pantalla en Android

Se puede concluir, que si se desarrolla en una baja densidad o en un tamaño pequeño de pantalla, no se llegará ni al 3% del mercado de Android.

2.2.2 Arquitectura en Android

Aplicaciones

Android se distribuye libremente con un conjunto de aplicaciones básicas incluyendo cliente de correo, envío de **SMS**, calendario, mapas, navegador y agenda de contactos entre otros. Todas las aplicaciones están escritas en Java. También, opcionalmente, puede venir con aplicaciones que no necesariamente han tenido que ser desarrolladas por **Google**.

Framework de aplicación:

Android permite construir aplicaciones variadas e innovadoras. Los desarrolladores pueden hacer uso libremente del *hardware* del dispositivo, ejecutar servicios en segundo plano, acceder a información de localización, añadir notificaciones a la barra de estado, establecer alarmas y otras muchas opciones.

Todas las aplicaciones instaladas en el dispositivo son equivalentes para el sistema operativo, todas tienen acceso a los mismos recursos y acceden a la misma **API**.

Con la idea de facilitar la reutilización de componentes, se permite especificar a cada aplicación un conjunto de interfaces con las que poder interactuar con el resto. De esta manera es posible usar unas aplicaciones dentro de otras. Por ejemplo, la aplicación oficial para ver la galería fotos tiene una opción para compartir las fotos a través de múltiples redes sociales.

El *Framework* de aplicación (entre otros) se compone de:

- **Sistema de vistas**: pueden ser utilizadas para construir aplicaciones incluyendo listas, rejillas (*grids*), cajas de texto, botones o, por ejemplo, un navegador embebido.

- **Proveedores de contenidos**: permiten a las aplicaciones acceder a datos de otras aplicaciones o compartir los suyos propios.

- **Administrador de notificaciones**: permite a todas las aplicaciones mostrar alertas en la barra de estado.

- **Administrador de actividades:** se encarga de conducir el ciclo de vida de las aplicaciones y de proveer una pila para poder navegar entre actividades.

Para ver la arquitectura de una manera más organizada de forma gráfica, se puede consultar la Figura 9 (17).

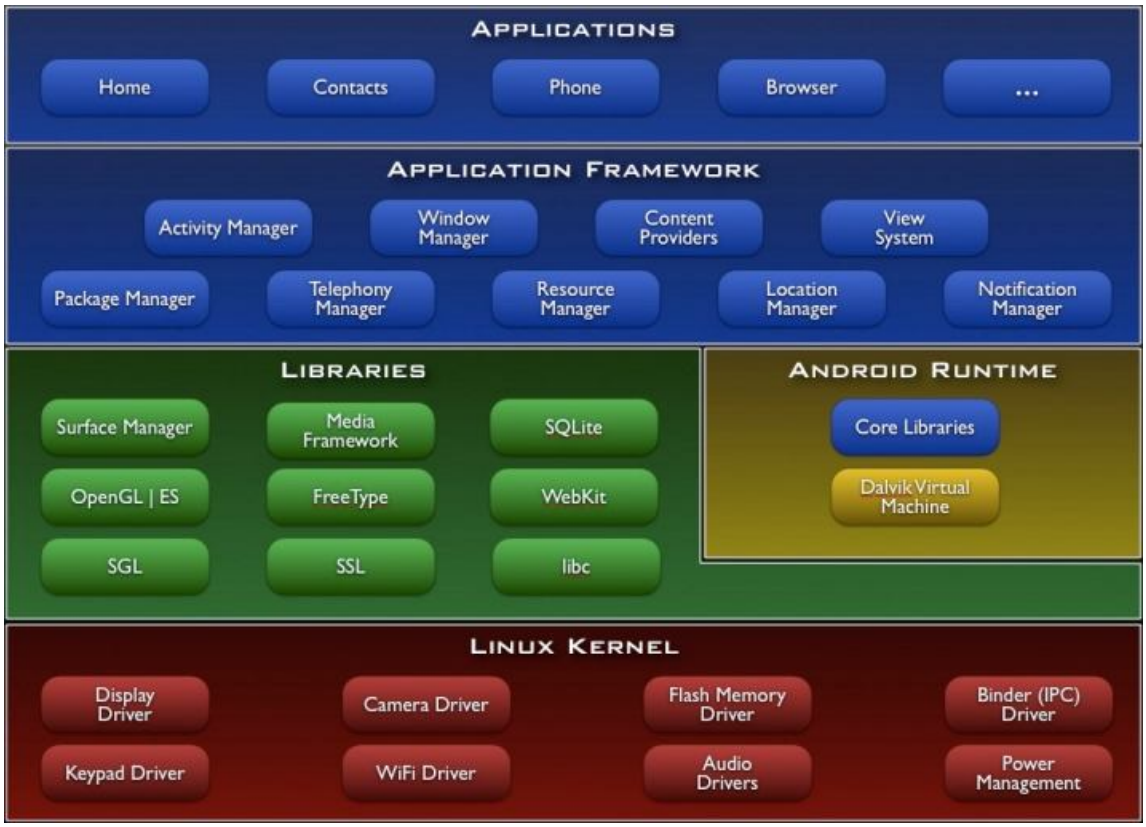


Figura 9: Arquitectura de Android (17)

Bibliotecas

Android incluye un conjunto de bibliotecas de **C/C++** que son usadas en varias partes del sistema. Su funcionalidad queda, por tanto, disponible para el uso de desarrolladores a través del uso del *framework* de Android. Algunas de las bibliotecas son las siguientes:

- **Biblioteca de *System C***: Una implementación tipo **BSD** de la biblioteca estándar de **C** (*libc*), adaptada para su uso en sistemas Linux embebidos.

- **Bibliotecas multimedia**: Se basan en la tecnología **OpenCORE** de la empresa *Packetvideo8*, estas bibliotecas soportan la reproducción y grabación de audio y vídeo, así como formatos de imágenes, de parte de los formatos más populares incluyendo **MPEG4, H.264, MP3, AAC, AMR, JPG y PNG**.

- **Administrador gráfico**: Encargado de gestionar el acceso al subsistema gráfico por las distintas aplicaciones y utilizar los gráficos 2D y 3D mediante capas.

- **LibWebCore**: Un motor que es parte tanto del navegador principal como de los embebidos.

- **SGL**: El motor de gráficos 2D.

- **Bibliotecas 3D**: Basadas en las **APIs** de *OpenGL ES*, versión 1.0.

- **FreeType**: Usada para el renderizado de fuentes vectoriales o en mapas de bits.

- **SQLite**: Un motor de bases de datos relacional ligero y potente disponible para todas las aplicaciones.

Android en tiempo de ejecución

Cada aplicación Android se ejecuta como un proceso, con su propia instancia de la máquina virtual *Dalvik*. Ésta ha sido diseñada para poder ejecutar múltiples máquinas virtuales de manera eficiente, haciendo uso del formato ejecutable *Dalvik* (.dex) que está optimizado para un mínimo uso de memoria.

La máquina virtual *Dalvik* descansa sobre el núcleo Linux que se encargará de la funcionalidad a bajo nivel relacionada con hilos y manejo de memoria a bajo nivel.

Kernel Linux

Android utiliza una versión 2.6 de Linux para sus funciones principales como seguridad, administración de la memoria, administración de procesos, pila de red y drivers. El *kernel*, además, funciona como abstracción entre el hardware y el resto de la pila de software.

2.2.3 Componentes de una aplicación Android

Google dispone de versiones del **SDK** para **Windows, Mac OS y GNU/Linux**, su uso recomendado es a través del **IDE** Eclipse si bien se pueden utilizar otras plataformas. El manager de emuladores y **SDK** puede verse en la figura Figura 10.

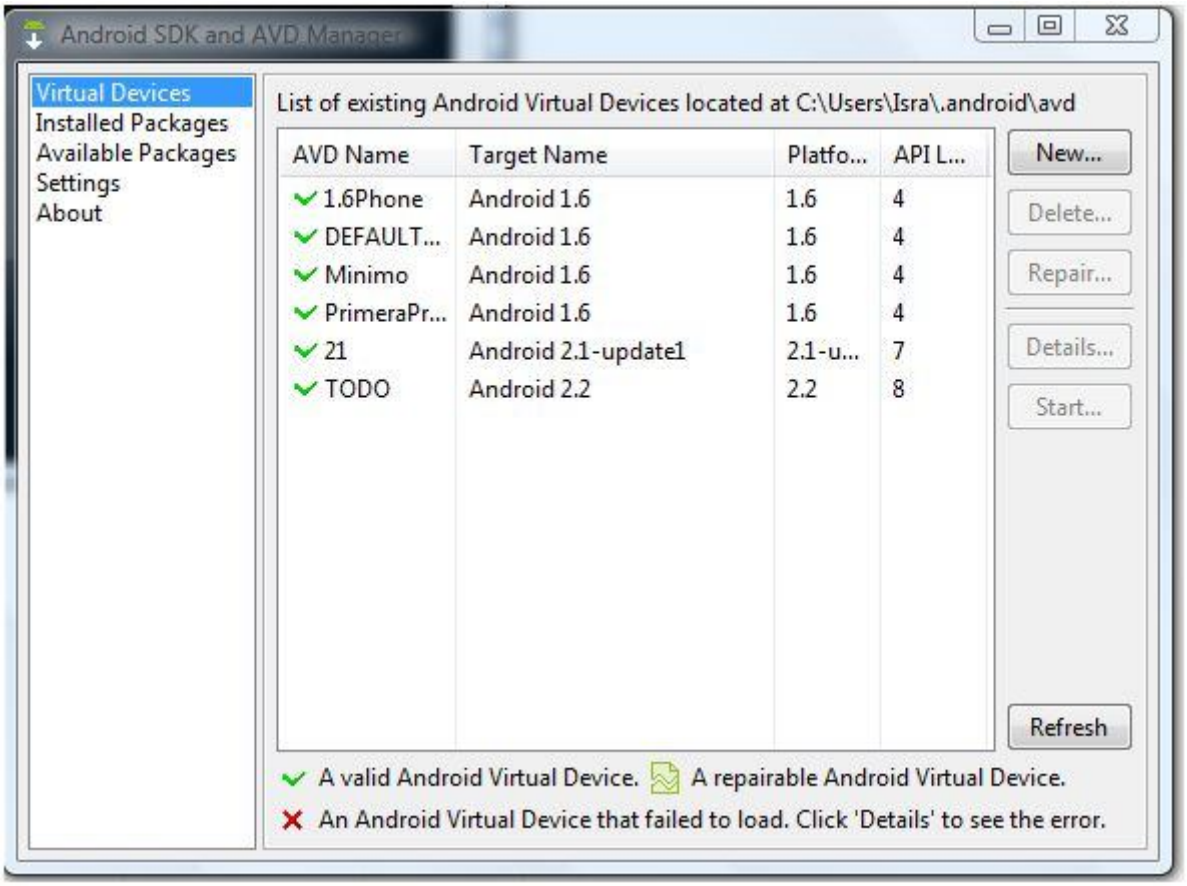


Figura 10: Android SDK y AVD Manager empleadas en este proyecto

El primer paso para el desarrollador es conocer los principales componentes de una aplicación, que son los que siguen a continuación.

Actividad (Activity)

Se trata de un componente que representa la funcionalidad completa de un programa en Android. Por ejemplo, una aplicación simple como la carpeta de mensajes puede estar dividida en varias actividades, una para la creación de nuevos mensajes, otro para ver los mensajes ya existentes, otra para la creación de mensajes multimedia, etc.

Intentos (Intents)

Disparan acciones predefinidas al activarse eventos externos. Por ejemplo, una aplicación podría estar registrada a la escucha de intentos para que al llegar un correo se active una alarma. Estos intentos no son sólo recibidos por las aplicaciones y lanzados por el sistema, sino que pueden provocarse desde nuestras aplicaciones y recogidas por otras.

Servicios (Services)

Se trata de tareas que se desarrollan en segundo plano, lo que significa que el usuario podrá iniciar una aplicación mediante una actividad y el servicio se mantendrá funcionando. Es el ejemplo más claro de multitarea.

Proveedor de contenidos (Content Provider)

Es el componente que se encarga de compartir datos entre procesos y aplicaciones. Se definen los parámetros en uno de los ficheros necesarios en todas las aplicaciones de Android, AndroidManifest.xml, que se encuentra en el directorio raíz del proyecto. En él se especifican sus componentes, los intentos que provee, se establecen los permisos que se necesitan para usar el hardware y otras propiedades de la aplicación.

2.2.4 Ciclo de vida de una aplicación

Como se ha dicho con anterioridad, en Android cada aplicación se ejecuta en su propio proceso debido, principalmente, a motivos de seguridad y protección ante errores de memoria, pero también hace que el sistema tenga que gestionar la creación y destrucción de una manera ordenada y clara de dichos procesos.

El sistema reacciona ante los cambios de aplicaciones y actividades que realiza el usuario almacenando las actividades que son utilizadas en una pila, de esta manera podemos deshacer nuestras acciones y volver a la actividad anterior sin más que pulsar uno de los cuatro botones que tienen todos los teléfonos Android, la tecla de volver.

El ciclo de vida de una actividad (18) se muestra en la Figura 11 y se compone de los siguientes estados:

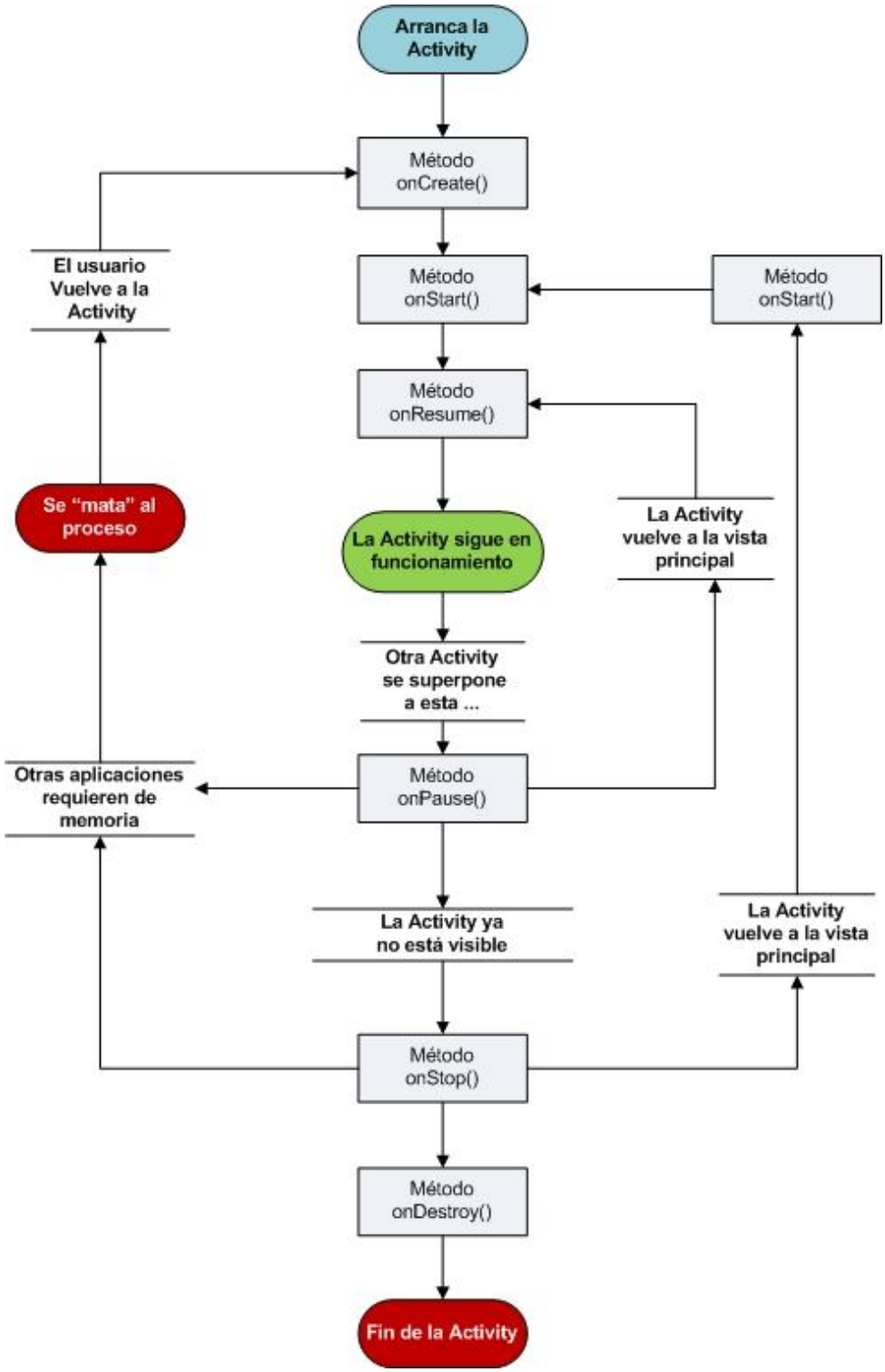


Figura 11: Ciclo de vida de una Activity en Android

Activo

La actividad se encuentra en este estado si está visible en la pantalla (estará en la parte superior de la pila). La actividad tiene, por tanto, el foco para las acciones del usuario.

En pausa

Una actividad que se encuentre visible pero que haya perdido el foco se encontrará en este estado. Para que esto ocurra, dicha actividad tiene que estar parcialmente oculta por otra. Las actividades en estado pausado se encuentran activas pero podrían ser terminadas por el sistema en caso de que el mismo se encontrase con muy poca memoria disponible.

Parado

La Actividad se encuentra parada si está por completo oculta de la pantalla. A pesar de ello, guardará toda la información de la que estaba haciendo uso, pero al no ser visible, es muy probable que el sistema la cierre cuando se necesite memoria.

2.3 Introducción a Ice (ZeroC)

ICE es un middleware con licencia **GPL**, es decir, libre, desarrollado por **ZeroC** y cuya finalidad es la de implementar una versión reducida y más simple de **CORBA** (19).

El objetivo de **ICE** es ofrecer al usuario una capa de *middleware* capaz de servir de nexo entre dos máquinas, y en la cual se pueda desplegar una aplicación orientada a objetos consiguiendo así que una máquina acceda a métodos implementados en otra. Todo ello bajo un escenario distribuido donde las máquinas entre sí puedan acceder mediante llamadas a métodos remotos.

La manera de hacer esto posible, es que una de las máquinas, tenga acceso a una interfaz donde estén descritos los métodos a los cuales se le puede acceder. Esta interfaz, debe ser genérica o dicho de otra manera, ambas partes han de entender estén programadas en el lenguaje que estén. Esta es una de las grandes virtudes de **ICE** y **CORBA**, que hacen que un sistema pueda crecer independientemente en la plataforma en la que esté desarrollado.

Para poder llegar a un sistema así de versátil, existen muchas otras tecnologías de *middleware* distintas. Entre otras las que están relacionadas en la Tabla 4.

Tecnología	Inconveniente
RMI (Remote Method Invocation)	Es dependiente de su lenguaje (Java)
WCF (Windows Communication Foundation)	Es dependiente de su plataforma (Windows)
XML-RPC (eXtended Markup Languaje – Remote Procedure Call) y SOAP (Simple Object Access Protocol)	No están diseñadas para aplicaciones de propósito general, solo para aplicaciones de servicio web o comúnmente conocidas como “Web Services”
CORBA(Common Object Request Broker Arquitechture)	Aunque cumple con los requisitos, es altamente complejo

Tabla 4: Tecnologías de middleware actuales.

Como se puede ver la Tabla 4, es por esto mismo por lo que surge **ICE**. Sin depender del lenguaje ni de la plataforma, y con la ventaja de tener una mayor facilidad a la hora de implementar una solución final, lo que **ICE** trata de cumplir es:

- Proveer un middleware orientado a objetos sin depender de plataforma y lenguaje.
- Facilitar la interconexión de sistemas heterogéneos de una manera más sencilla, favoreciendo el uso y la implantación en el mercado de ella así como facilidad de su aprendizaje.
- Eficiencia en lo referente a uso de ancho de banda y uso de **CPU**
- Una plataforma segura desde una primera instancia sin necesidad de parches.

A continuación se presentan algunas de las características principales de **ICE**. Como se ha comentado, **ICE** puede hacer que los datos intercambiados sean independientes de plataforma y lenguaje. Para que esta interconexión e intercambio de datos bajo esas premisas sea posible, **ICE** usa un lenguaje, **SLICE** (*Specification Language for ICE*), al igual que **CORBA** utiliza **IDL**.

En el modo asíncrono de **ICE**, el cliente vuelve a la ejecución normal sin quedarse bloqueado, ya que implementa un parámetro de "*callback*" que permite que el hilo vuelva directamente a la ejecución sin quedarse esperando. Esto lo hace recogiendo el resultado de la llamada al método que ha hecho, cuando se le avisa mediante una función de "*callback*".

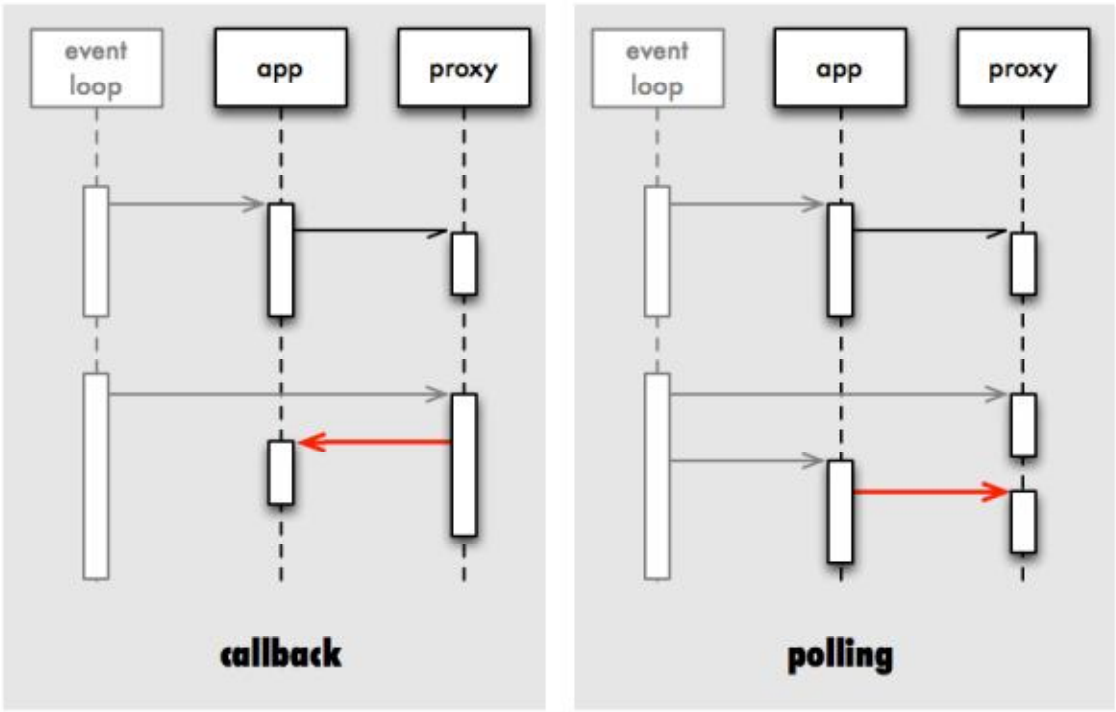


Figura 12: Representación gráfica de una función de *callback* y una de *polling*.

Una de las grandes virtudes de la tecnología **ICE**, es que provee una gran cantidad de servicios los cuales son añadidos al núcleo central de **ICE**. Esto supone una gran comodidad para el desarrollador, que puede acceder a nuevas funcionalidades de manera separada y que no tiene el núcleo de **ICE** por sí mismo. Estos servicios son, ***IceUtil, IceBox, IceSSL, IceGrid, IceStorm, Freeze, FreezeScript, IcePath2*** y ***Glacier2*****.**

Servicio	Explicación
IceUtil	Manipulación de hilos sólo disponible en C++
IceBox	Servidor de aplicaciones creado para *ICE*
IceSSL	Cifrado, autenticación e integridad mediante mensajes SSL.
IceGrid	Servicio para implementar y activar arquitecturas de tipo *GRID*.
IceStorm	Servicio de publicación-subscripción y federación de tópicos del sistema mediante *ICE*.
Freeze	Servicio de persistencia.
FreezeScript	Herramienta de manejo de *Freeze.*
Glacier2	Sistema de seguridad que facilita el desarrollo de sistemas distribuidos con *firewalls*.
IcePath2	Servicio del control de versiones de ICE.

Tabla 5: Servicios que provee ICE.

2.3.1 Lenguajes de definición de tipos

En esta sección se presentarán las diferencias entre el lenguaje de definición de tipos que utiliza **CORBA**, **IDL** (*Interface Definition Language*), y el que utiliza **ICE**, **SLICE** (*Specification Language for ICE*).

El objetivo de de los lenguajes de definición de tipos es servir de diccionario entre lenguajes diferentes ofreciendo una interfaz común a cualquier lenguaje soportado. De modo que si se tiene un cliente y un servidor desarrollados en dos lenguajes diferentes y en plataformas diferentes, usando un interfaz común, puedan comunicarse.

Es decir, estos lenguajes permiten que ambas tecnologías sean independientes del lenguaje, aumentando la escalabilidad del sistema como se ha comentado anteriormente. Actualmente **SLICE soporta** el mapeo a los siguientes lenguajes: ***C++, Java, C#, Python, PHP, Ruby***, **.NET** y más recientemente para aplicaciones sobre el iOS de Apple, **Objetive-C** mientras que **IDL** permite mapeo a ***C, C++, Java, Smalltalk, COBOL, Ada, Lisp, PL/1, Python***.

Como diferencias de tipos entre **SLICE** e **IDL**, se pueden encontrar las siguientes:

- **IDL** no tiene soporte para mapear diccionarios, **SLICE** sí. Esto supondría en **CORBA** una complejidad añadida a la hora, por ejemplo, de enviar una "*hash-table*" de Java.

- **SLICE** sólo permite una manera de implementar, al contrario que **IDL**, añadiendo a **CORBA** la complejidad de poder implementar un sistema de distintas maneras.

- Al contrario que en **IDL**, **SLICE** permite incluir metadatos en cada mensaje enviado, de manera que la estructura de los datos enviados, se mantenga igual que lo que haya sido negociado, pero exista la posibilidad de añadir información adicional.

Se puede apreciar, que **SLICE** es una representación más eficiente que **IDL** y si se ve el código realizado en **SLICE**, se confirma esa primera apreciación. No con esto se quiere decir que usar **IDL** sea un inconveniente, ya que ambos lenguajes funcionan a la perfección para cada tecnología usada. Aun así, si en el sistema prima la sencillez y la eficiencia en un intervalo más corto de desarrollo, **SLICE** sería la opción a tomar.

2.3.2 ICE for Android

ICE for Android, lanzado a finales de diciembre del 2008 (20), es una adaptación de **ICE** para sistemas Android el cual puede ser descargado de su página web (21)**.**

Las funcionalidades en base son las mismas a los demás sistemas, pero adaptadas a las exigencias de los dispositivos Android. Se puede usar a partir de la versión de Android 1.5 y no incluye los servicios **ICE** que están descritos en la Tabla 5.

El desarrollo de **ICE** for Android se desarrolla paralelamente a ICE, pero de manera más lenta. En dos años ha sacado dos versiones, siendo la última la versión **0.1.1** (22), perfectamente funcional con la penúltima versión de **ICE 3.3.1** dado que las mejoras y los cambios de **ICE 3.4.1**, aunque no influyen para el uso de **ICE** for Android, éste no ha sido actualizado.

2.3.3 Introducción a Java Media Framework 2.1.1 JMF

El **API JMF** (23)**,** es una definición del interfaz utilizado por Java para la utilización de multimedia y su presentación en *applets* y aplicaciones. Como definición de interfaz, **JMF** no tiene por qué proporcionar las clases finales que manejan los datos multimedia, pero dice cómo los proveedores de dichas clases deben encapsularlas y registrarlas en el sistema.

Históricamente (24), **JMF** nació con el objetivo de poder representar datos multimedia en los programas pero, en la última versión (**JMF** 2.0.) las capacidades se extienden a todo tipo de tratamiento como la adquisición, procesado y almacenaje de datos multimedia, así como la transmisión y recepción a través de la red mediante el protocolo **RTP**.

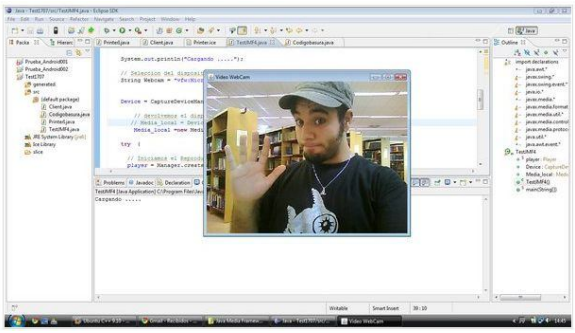


Figura 13: Imagen tomada con las librerías JMF

JMF 2.0. ha sido desarrollada por *Sun e IBM* y no está incluida en las especificaciones de Java 2 o de la máquina virtual, por lo que es necesario obtener el paquete adicional que contiene el **JMF** para la plataforma que se esté utilizando.

Los formatos que acepta, excluyendo MP3 para toda plataforma que no sea Windows son:

- **Protocolos:** FILE, HTTP, FTP, RTP
- **Audio:** AIFF, AU, AVI, GSM, MIDI, MP2, MP3, QT, RMF, WAV
- **Video:** AVI, MPEG-1, QT, H.261, H.263
- **Otros:** HotMedia.

La versión **2.1.1** de **JMF** incluye:

- **RTP API** mejorado
- Soporte para H.263/1998 (**RFC** 2429) para poder interoperar con servidores **RTSP** basados en *Darwin*.
- Renderizador de audio directo y capturador
- Mayor eficiencia con compiladores de Java nuevos.
- Aplicación llamada "**JMF** Customizer" añadida a todas las versiones de **JMF**.
- Soporte para **HTTPS**, **JAWT** y *"**UNC** pathnames"*
- Optimizaciones y corrección de fallos frente a las versiones anteriores.

Capítulo 3

Estudio del protocolo ICE

En este capítulo, se tratará de profundizar un poco más en la capa de middleware de comunicaciones **ICE** en distintos escenarios. Para probar aún más de manera realista lo que **ICE** puede llegar a hacer, se desarrolló una aplicación entre Linux y Windows en Java, consistente en peticiones recursivas de imágenes a un servidor, para luego mostrarlas secuencialmente y dar una apariencia de video. También se realizaron pruebas entre distintos lenguajes de programación como **C++** y Java.

3.1 El protocolo ICE

En este apartado, podemos centrar el estudio del protocolo, en la definición que nos proporciona **ZeroC** en su manual del protocolo **ICE** (25) el consiste en 3 partes importantes:

- Un conjunto de reglas de codificación que determinan cómo se van a serializar los diferentes tipos de datos que soporta.
- Un número de mensajes que son intercambiados entre el cliente y el servidor, junto con reglas de actuación en función de la circunstancia.
- Un conjunto de reglas que determinan cómo el cliente y el servidor se ponen de acuerdo en una versión de protocolo y codificación.

Se explicarán dichas reglas de codificación y los tipos de mensajes así como información general del protocolo.

3.1.1 Codificación de datos

El protocolo **ICE,** el tamaño de los datos trata de hacer una compresión y codificación de los datos lo más eficiente posible eliminando bits de "*padding*" y representando cada tipo primitivo de datos con el número justo de bytes.

Tamaños

En **ICE** tamaño de los datos se codifica en función del "tamaño del tamaño". Esto quiere decir, que para codificar el tamaño de un *String*, por ejemplo, en función de las premisas que se comentan abajo, tendrá un tamaño u otro. Esto optimiza mucho la manera de codificar dado que cumple estas dos normas:

1. Si el número de bytes es menor de 255, el tamaño se codifica con un solo byte
2. Si el número de bytes es mayor o igual a 255, el tamaño se codifica con un byte de valor 255, seguido de un int indicando el número de bytes restantes.

Esta codificación ahorra mucho ancho de banda ya que si una aplicación envía cadenas cortas de texto, es significantemente más eficiente codificar el tamaño con un byte que siempre con un int.

No se va a entrar en mayores detalles en la forma que tiene **ICE** de codificar cada tipo de dato, pero puede verse en la Tabla 6 los tipos que soporta.

Tipo	Tamaño
bool	1 byte
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes (23 bits para mantisa, 8 exponente, 1 para signo)
double	8 bytes (52 bits para mantisa, 11 exponente, 1 para signo)
Strings	Según tamaño
secuences (arrays)	Según tamaño
dictionaries	Según tamaño
enumerators	En función del número de enumeradores: 1-127 se codifica como byte. 128-32767 se codifica como short. Más de 32767 se codifica como int.
struct	El tamaño de los miembros del struct en el orden que se hayan definido
exceptions	Según tamaño
Clases	Según tamaño
Interfaces	Según tamaño
Proxies	5 bytes + variables

Tabla 6: Codificación de datos en ICE

Si se quisiera ampliar información acerca de cómo el protocolo **ICE** codifica las estructuras de datos no tan triviales como los primitivos, se puede consultar el manual de **ICE** el cual está accesible en la página de **ZeroC** (26)**.**

Ejemplo de Marshaling

Para entender un poco mejor la codificación de los datos y cómo la compresión del tamaño puede afectar a la eficiencia del protocolo, se analizará el ejemplo descrito en el manual de **ICE** en el apartado **37.2.11** (27)

Definiendo la interfaz y las clases que se muestran en la Tabla 7, se podrá ver cómo al intercambiar cadenas de texto pequeñas, la eficiencia del protocolo aumenta con este tipo de compresión.

interface **SomeInterface** {
  void op1();
};
 	
class **Base** {
  int baseInt;
  void op2();
  string baseString;
};
	class **Derived** extends Base implements **SomeInterface**
	{
	  bool derivedBool;
	  string derivedString;
	  void op3();
	  double derivedDouble;
	};

Tabla 7: Ejemplo de Marshalling

Ahora, se tiene en cuenta que a la hora de codificar los datos, sólo se tienen en cuenta los datos que ocupan espacio, es decir, los métodos op1(), op2() y op3() no generan carga dado que no devuelven nada.

	Variables	Tipo	Valor	Valor codificado (bytes)
Primera instancia	baseInt	int	99	4
	baseString	string	"Hello"	6
	derivedBool	bool	true	1
	derivedString	string	"World!"	7
	derivedDouble	double	3.14	8
Segunda instancia	baseInt	int	115	4
	baseString	string	"Cave"	5
	derivedBool	bool	false	1
	derivedString	string	"Canem"	6
	derivedDouble	double	6.32	8

Tabla 8: Tamaño y valor de los datos codificados en el ejemplo

Teniendo claro esto, se ha realizado una prueba de dos instancias con los valores que se pueden ver en la Tabla 8. El tamaño final del paquete a enviar es de **124 bytes**, añadiendo todos los parámetros de control que se envían en cada transacción.

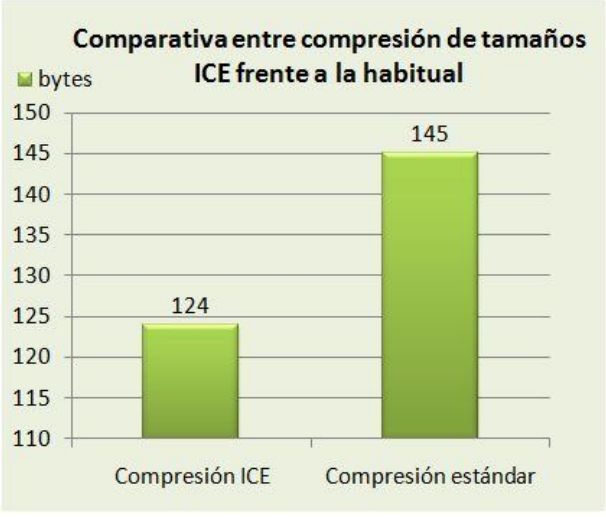


Figura 14: Comparativa entre compresión de tamaños ICE frente a la habitual

El total de 124 bytes se puede ver desglosado en la Tabla 9, marcadas las líneas involucradas en la definición de cadenas de texto, que es donde se aprecia la mejora.

Se tienen en total 7 cadenas de texto que no llegan a 254 bytes caracteres cada una, por lo que el *overhead* que introduce para indicar el tamaño del *string*, es de **7 bytes**. Si se codificara como se hace en la mayoría de protocolos (el tamaño suele estar representado con un int, 4 bytes), tendríamos de *overhead* **28 bytes** frente a esos **7**. El resultado obtenido se puede ver gráficamente en la Figura 14.

Valor codificado	Tamaño in Bytes	Type	Byte offset
1 *(identity)*	4	int	0
0 *(marker for class type ID)*	1	bool	4
"::Derived" *(class type ID)*	10	**string**	5
20 *(byte count for slice)*	4	int	15
1 *(derivedBool)*	1	bool	19
"World!"*(derivedString)*	7	**string**	20
3.14 *(derivedDouble)*	8	double	27
0 *(marker for class type ID)*	1	bool	35
"::Base" *(type ID)*	7	**string**	36
14 *(byte count for slice)*	4	int	43
99 *(baseInt)*	4	int	47
"Hello" *(baseString)*	6	**string**	51
0 *(marker for class type ID)*	1	bool	57
"::Ice::Object" *(class type ID)*	14	**string**	58
5 *(byte count for slice)*	4	int	72
0 *(number of dictionary entries)*	1	size	76
2 *(identity)*	4	int	77
1 *(marker for class type ID)*	1	bool	81
1 *(class type ID)*	1	size	82
19 *(byte count for slice)*	4	int	83
0 *(derivedBool)*	1	bool	87
"Canem"*(derivedString)*	6	**string**	88
6.32 *(derivedDouble)*	8	double	94
1 *(marker for class type ID)*	1	bool	102
2 *(class type ID)*	1	size	103
13 *(byte count for slice)*	4	int	104
115 *(baseInt)*	4	int	108
"Cave" *(baseString)*	5	**string**	112
1 *(marker for class type ID)*	1	bool	117
3 *(class type ID)*	1	size	118
5 *(byte count for slice)*	4	int	119
0 *(number of dictionary entries)*	1	size	123
		Total:	124

Tabla 9: Tamaño completo de los datos enviados

3.1.2 Cabeceras y tipos de mensaje

Los tipos de mensaje en **ICE** son los siguientes:

- *Request* (De cliente a servidor)
- *Batch Request* (De cliente a servidor)
- *Reply* (De servidor a cliente)
- *Validate connection* (De servidor a cliente)
- *Close connection* (Ambas direcciones)

La descripción de un mensaje **ICE**, se compone de cabecera y cuerpo de mensaje a excepción de los casos de *Validate connection* y *Close connection*.

La cabecera tiene el siguiente formato:

Campo	Tamaño	Descripción
magic	int	Número "mágico" codificado en ASCII con los valores: **'I', 'c', 'e', 'P'** (0x49, 0x63, 0x65, 0x50)
protocolMajor	byte	"*Major version*" del protocolo
protocolMinor	byte	"*Minor version*" del protocolo
encodingMajor	byte	"*Major version*" de la codificación
encodingMinor	byte	"*Minor version*" de la codificación
messageType	byte	Tipo de mensaje
compressionStatus	byte	Indica si está comprimido o no el mensaje
messageSize	int	El tamaño del mensaje, incluida cabecera.

Tabla 10: Campos de la cabecera del protocolo ICE

El campo de *messageType*, es el que indica qué mensaje es el recibido o enviado, y sigue la equivalencia:

- **(Valor 0) Request**
- **(Valor 1) Batch Request**
- **(Valor 2) Reply**
- **(Valor 3) Validate connection**
- **(Valor 4) Close connection**

Posibles valores de messageType

Se comentan más adelante en detalle los cuerpos de cada uno de los mensajes posibles en **ICE**.

Request

Los mensajes de tipo *Request* tienen el siguiente formato:

Campo	Descripción
requestId	ID de petición
id	ID de objeto
facet	Nombre de la faceta (Secuencia de uno o ningún elemento)
operation	Nombre de la operación
mode	Representación de : Ice::OperationMode (0=normal,2=idempotente)
context	Contexto de invocación
params	Parámetros encapsulados en orden de aparición.

Tabla 11: Descripción de los campos de un mensaje de tipo Request

Batch Request

Los mensajes de tipo *Batch Request* tienen el siguiente formato:

Campo	Descripción
id	ID de objeto
facet	Nombre de la faceta (Secuencia de uno o ningún elemento)
operation	Nombre de la operación
mode	Representación de :Ice::OperationMode
context	Contexto de invocación
params	Parámetros encapsulados en orden de aparición.

Tabla 12: Descripción de los campos de un mensaje de tipo Batch Request

Destacar que no se requiere en este tipo de respuesta el campo `requestID`, ya que es una petición de respuesta porque sólo puede estar en un escenario de una dirección.

Reply

Para los mensajes de *Reply*, tienen el siguiente formato:

Campo	Descripción
requestID	ID de petición
replyStatus	Estado de la respuesta
body	Cuerpo que varía en función del estado de respuesta.

Tabla 13: Descripción de los campos de un mensaje de tipo Reply

El campo del estado de la respuesta, podrá obtener los siguientes valores mostrados en la lista.

- **(Valor 0) Success**
- **(Valor 1) User Exception**
- **(Valor 2) Object does not exist**
- **(Valor 3) Facet does not exist**
- **(Valor 4) Operation does not exist**
- **(Valor 5) Unknown Ice local exception**
- **(Valor 6) Unknown Ice user exception**
- **(Valor 7) Unknown exception**

Tipos de mensajes *Reply*

En función a los valores recibidos, el cuerpo del mensaje variará en función de la respuesta generada. En los valores **1, 3, 4, 5, 6, 7**, el resultado siempre será el mismo; el cuerpo del mensaje será una cadena de texto, indicando en cada caso el problema generado.

Entonces, en el caso de valor **0**, cuando la operación es correcta, en el cuerpo del mensaje, irán los parámetros solicitados, y en el caso de **2**, el mensaje cambia radicalmente con la forma que sigue a continuación.

Campo	Descripción
id	ID del objeto
facet	Estado de la respuesta
operation	El nombre de la operación que ha causado el error.

Tabla 14: Descripción de los campos del tipo de Reply número 2

Validate connection y Close connection

Ambos mensajes son para el control de la conexión **ICE** entre los dos entes que participan. En el caso de **Validate**, se envía cuando el servidor recibe una nueva conexión y esta es posible, y **Close** se genera cuando se aborta la conexión. El cometido de estos dos mensajes se puede ver en la máquina de estados del protocolo **ICE** en la Figura 15.

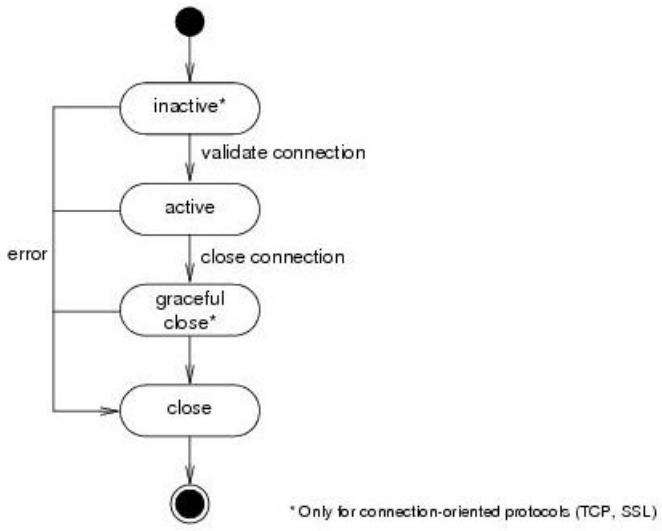


Figura 15: Diagrama de estados de la conexión de ICE

3.2 Análisis de distintos escenarios

Las pruebas para esta parte del proyecto, fueron realizadas bajo el simulador de máquinas virtuales ***VMware Workstation®*** (28) en su versión **6.5.2** en la cual se instaló la versión de ***Ubuntu 9.10*** (29)

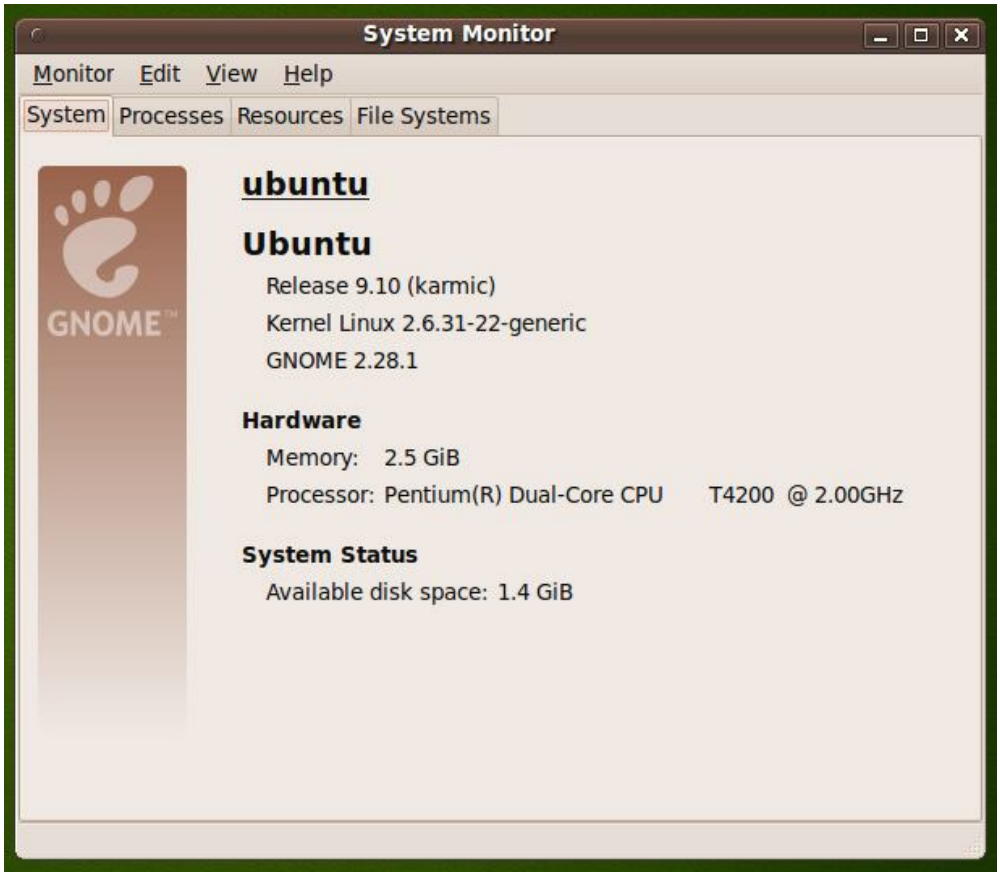


Figura 16: Distribución de Ubuntu instalada para el proyecto en la máquina virtual

El resto del desarrollo del proyecto se realizó en un ordenador portátil ***Acer Extensa 5635ZG***, con ***Windows Vista SP1 de 32 bits***. Sus especificaciones pueden verse en la Figura 17. Aunque en el desarrollo de este proyecto se han utilizado 3 sistemas operativos distintos, es posible realizarlo en muchos más como *Solaris, Mac OSx* y varias distribuciones de *Linux*.

Figura 17: Especificaciones de la máquina *Windows*

3.2.1 Librerías empleadas y entornos de trabajo

En la Tabla 15 se resume las versiones de **ICE** y **ICE** para *Android* utilizadas en los diferentes escenarios más relevantes para probar y familiarizarse con las funcionalidades de **ICE**.

Aplicación (Lenguaje)	Cliente	Servidor
Prueba de protocolo ICE (Java/Java)	Linux , **ICE** 3.4.1	Windows, **ICE** 3.3.1
Prueba de protocolo ICE (C++/Java)	Linux , **ICE** 3.4.1	Windows, **ICE** 3.3.1
Aplicación de video (Java)	Linux, **ICE** 3.4.1	Windows, **ICE** 3.3.1
SharePhoot(Java/Java)	Android 1.6	Windows **ICE** 3.3.1

Tabla 15: Escenarios probados para la fase de estudio.

En lo referente a la versión de Java utilizada durante todo el proyecto, ha sido la ***build 1.6.0_17-b04*** y para C++, ***4.4.1-ubuntu9***.

Se utilizaron adicionalmente, la *release* **JMF 2.2.1** (*Java Media Framework*) en Windows para la implementación de la aplicación de toma de imágenes a través de cualquier dispositivo de video. Se desarrollaron los trabajos de Java siempre en entornos ***Eclipse Ganymede 3.4.x y Galileo 3.5.x*** y los de **C++** directamente en ***gedit 2.28.0***.

Problemas con ICE para Android

ZeroC nos provee de las librerías de **ICE** para programar en el lenguaje que se quiera pero a la hora de programar para Android, ha de hacerse en Java, aunque ése no es el problema. El problema llega a la hora de incluir las librerías de la última versión disponible de **ICE** durante el desarrollo de este proyecto con las librerías generadas para Android.

Se hicieron varias consultas en el foro de **ZeroC** a los creadores de la tecnología, y se llegó a la conclusión que las versiones generadas de las librerías para Android (más concretamente, la versión **0.1.1**), no eran compatibles con las versiones posteriores, sólo se podían hacer funcionar bajo la distribución **3.3.1** de **ICE**, por lo que se tuvo que reinstalar todo el sistema para poder desarrollar en Android apropiadamente.

3.2.2 Funcionamiento básico de ICE

El funcionamiento de la conexión **ICE**, se explicará en código Java y está basado en el ejemplo más básico que se puede encontrar en el manual de usuario de **ICE 3.x** (26)

Como en toda conexión de red, si un cliente quiere acceder a un servidor, éste primero ha de estar ejecutándose para poder atender sus peticiones. Si no, a no ser que implemente algún mecanismo de detección de conexión no válida, el cliente puede quedarse congelado en el proceso de intentar conectarse.

Aunque se intentará introducir el menor código posible, no es factible una explicación de cómo funciona **ICE** a nivel de implementación sin una mínima parte de código.

La explicación abajo escrita, corresponde con la manera que implementa la aplicación **SharePhoot** la conexión de **ICE**, ya que se ha considerado que es el ejemplo más y más completo. No obstante, puntualizar que los ejemplos descritos en el apartado 3.2.1, su implementación dista poco de la explicada a continuación. Sólo cambian los valores de los parámetros y los nombres de las variables de red.

Servidor:

El primer paso que hay que dar, es inicializar el objeto *Communicator* de **ICE**. Se le pueden incluir parámetros opcionales, pero para el escenario representado, es suficiente.

```
ic = Ice.Util.initialize(args);
```

Después, se crea un adaptador **ICE** el cual se encarga de establecer las opciones de la conexión tales como protocolo, puerto etc.

"*ImageServerBind*" será el identificador de red por el cual se le podrá buscar. Al no especificar ninguna dirección **IP**, se establecerá una escucha en todos los adaptadores de red que estén disponibles, así las peticiones pueden llegar por cualquier puerta de enlace.

En este caso, se ha optado por establecer una conexión **TCP** en el puerto **60000**, en todos los adaptadores, y con un *timeout* "infinito", ya que no se le ha especificado ningún tiempo.

```
Ice.ObjectAdapter adapter = ic.createObjectAdapterWithEndpoints
            ("ImageServerBind", "tcp -p 60000");
```

Se crea un nuevo objeto de tipo *ImageServerI*, el interfaz, por el cual se podrán acceder a todos métodos que remotamente el interfaz.

```
Ice.Object iceObject = new ImageServerI();
```

Al objeto *adapter*, solo le falta que se le añada la ID bajo la cual se ha registrado el servidor en la red, junto con el objeto *ImageServerI*. Así quedará listo el adaptador para ponerlo en marcha.

```
adapter.add(iceObject, ic.stringToIdentity("ImageServerBind"));
```

Llegados a este punto es cuando se activa el adaptador, se mantendrá en segundo plano y atenderá todas las peticiones por **TCP**, en el puerto 60000 y desde cualquier interfaz.

```
adapter.activate();
```

El objeto *Communicator* quedará a la escucha por si hay algún problema con la conexión o por si se le pide que deje de funcionar.

```
ic.waitForShutdown();
```

Hasta aquí se explica cómo arrancar un servidor **ICE**, ergo ahora se explicará como acceder a los servicios que ofrece el servidor mediante el cliente.

Cliente:

Aunque la implementación del cliente es bastante parecida a la del servidor, difiere en dos cosas; que sí debe especificar una **IP** de destino y que en vez de crear un adaptador, se debe crear un objeto en local para realizar las peticiones remotas.

Al igual que en el servidor, se necesitará un objeto de tipo *Communicator*, y al cual se le inicializa de la misma forma.

```
Ice.Communicator ic = Ice.Util.initialize(args);
```

Con la siguiente línea, se creará el objeto *proxy* el cual proveerá la conectividad básica con el servidor. A este proxy se le deberán de indicar todos los parámetros de la comunicación con el servidor. En este caso, "*ImageServerBind*", es el identificador con el que el servidor está registrado en la red. A esto se le añade el protocolo (**TCP**), la dirección **IP** del servidor, así como un tiempo máximo de intento de conexión (un *timeout* en milisegundos). Finalmente, se le indica el puerto donde está escuchando el servidor, que como ya se ha dicho, es el 60000.

Hay que ser muy cuidadoso en este aspecto, dado que en función de la complejidad de la red, el valor del *timeout* ha de ser lo suficientemente grande para no expirar antes de poder establecer la conexión. En este caso es de **500**ms porque es una prueba de ámbito local.

```
Ice.ObjectPrx base = ic.stringToProxy("ImageServerBind:tcp -h "+ ip+ " -t
                                500 -p 60000");
```

Una vez quede creado e inicializado el objeto base, se procede a crear el objeto local que se encargará de realizar las llamadas al servidor. Aquí es donde reside la necesidad de tener una implementación **SLICE** en cada parte de la comunicación, para que existan objetos del tipo <*MODULO_ICE*.*INTERFAZ*> y poder instanciarlos en el ámbito local.

```
Sharephoot.ImageServerPrx tester = Sharephoot.ImageServerPrxHelper.
                            checkedCast(base);
```

Una vez creado el objeto de tipo *Sharephoot.ImageServerPrx*, aparte de llamar remotamente a todos los métodos que nos ofrece la interfaz remota, podemos realizar una prueba de conexión haciendo un *ping* a la máquina objetivo, la cual ya está definida dentro de este objeto.

```
tester.ice_ping();
```

En esta prueba del *ping* en particular, si lanza cualquier tipo de excepción, es el resultado negativo de la misma, sabiendo así que la conexión no es posible y se puede pasar a tratar dicha excepción de la manera que se necesite.

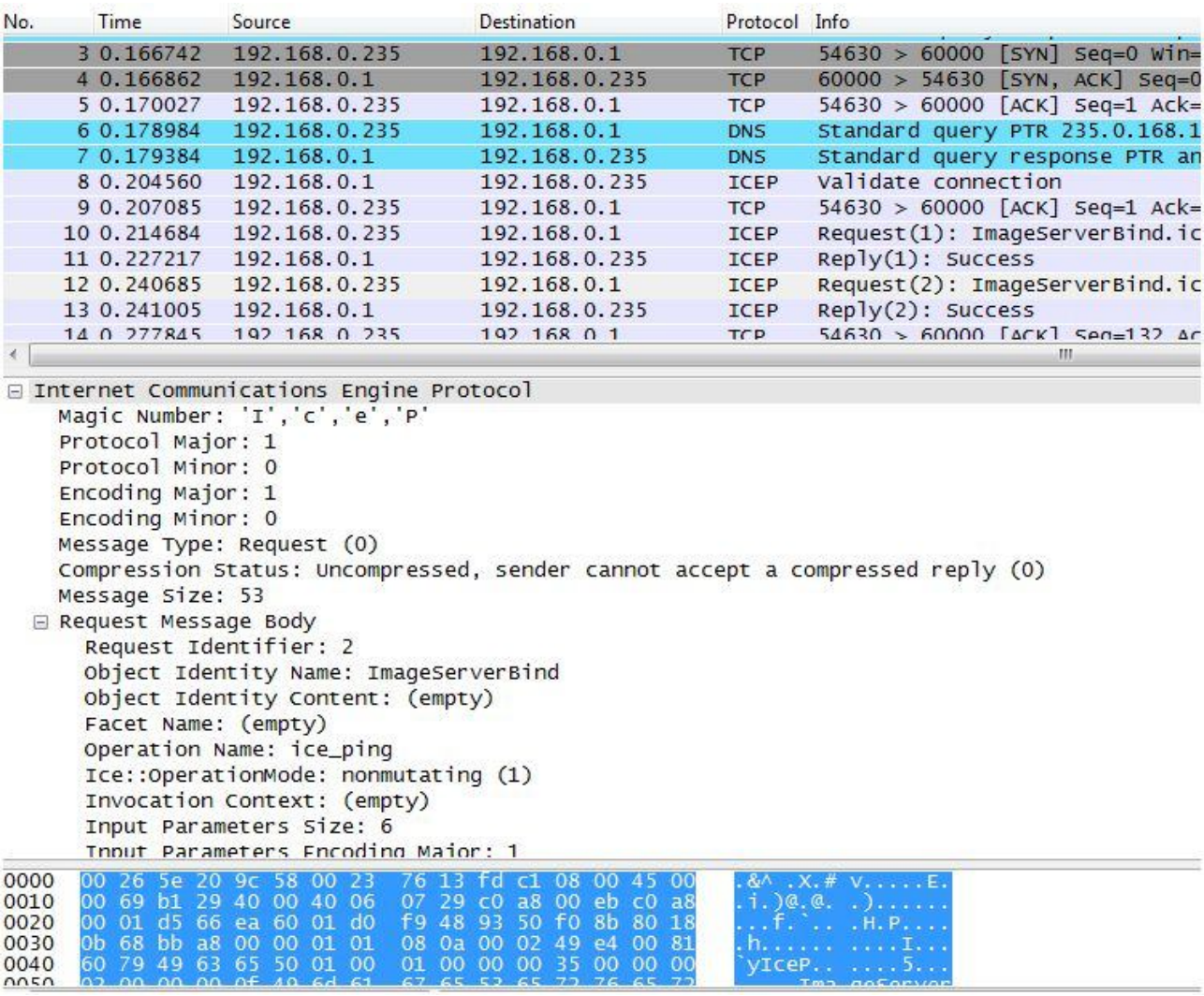


Figura 18: Captura del ice_ping() descrito en este apartado

En la Figura 18, se puede ver una captura realizada con el *sniffer* de red **Wireshark** (30), donde se muestran los mensajes intercambiados entre cliente y servidor en una prueba de conexión.

3.2.3 Prueba de escenarios

Las imágenes a continuación, están tomadas de capturas analizadas con **Wireshark**, y corresponden con las pruebas realizadas en los escenarios **Linux C++ - Windows Java y Linux Java – Windows Java.**

La conclusión a la que se quiso llegar, es que es indistinto el programar en un lenguaje u otro, ya que los datos empaquetados van a ser los mismos sea el lengua que sea.

Esto se puede comprobar en la Figura 19 y en la Figura 20 que el mensaje enviado a través del objeto **ICE** *SimplePrinter*: "*Mensaje molón para ICE*", los datos encapsulados en ambas pruebas ocupan 24 bytes indistintamente.

De hecho, se puede comprobar en las mismas capturas, que aunque se hayan hecho en momentos distintos, no es posible diferenciar en qué lenguaje están programadas ambas pruebas.

La Figura 21, es la respuesta que obtuvieron ambas peticiones del servidor.

Se puede concluir que la interfaz que ofrece el servidor, esté programada en el lenguaje en el que esté, el tratamiento de los datos siempre lo hará de la misma manera.

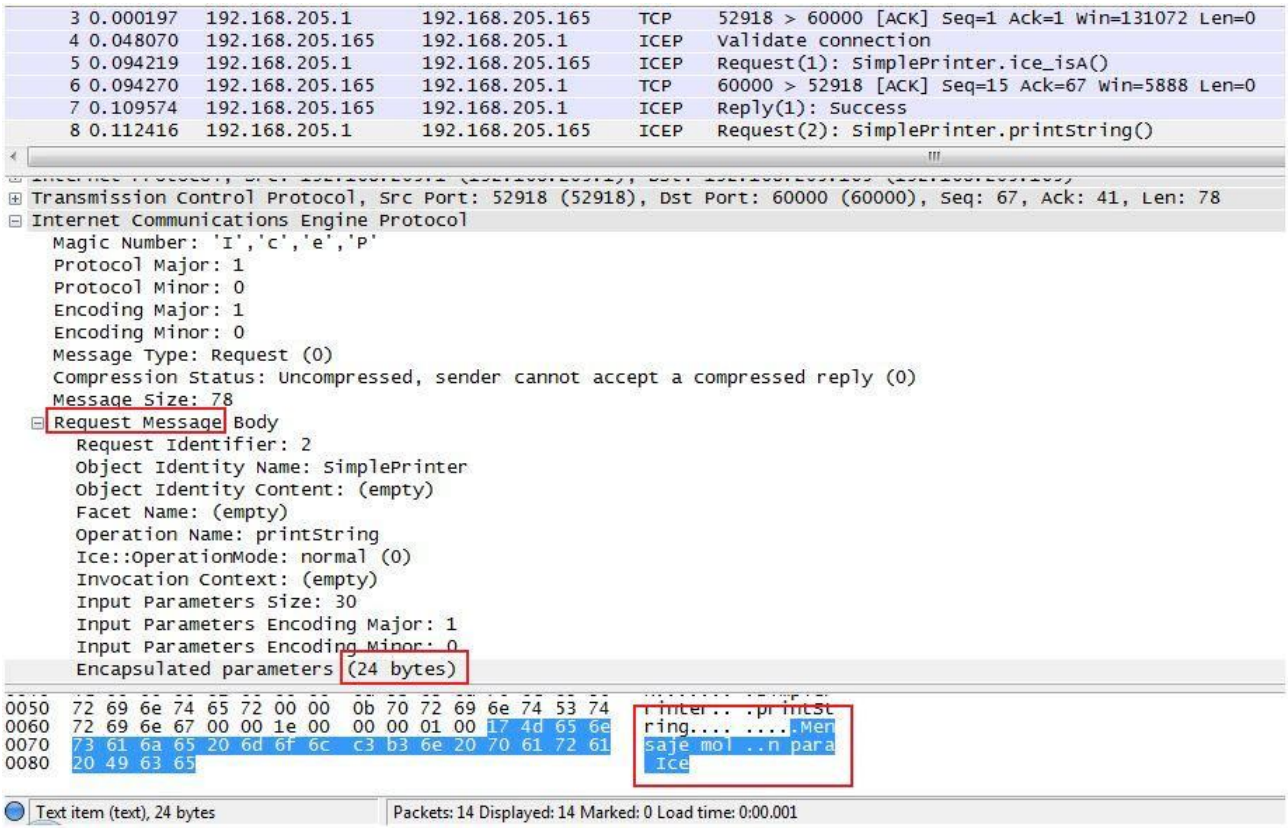


Figura 19: Captura del escenario Java - Java

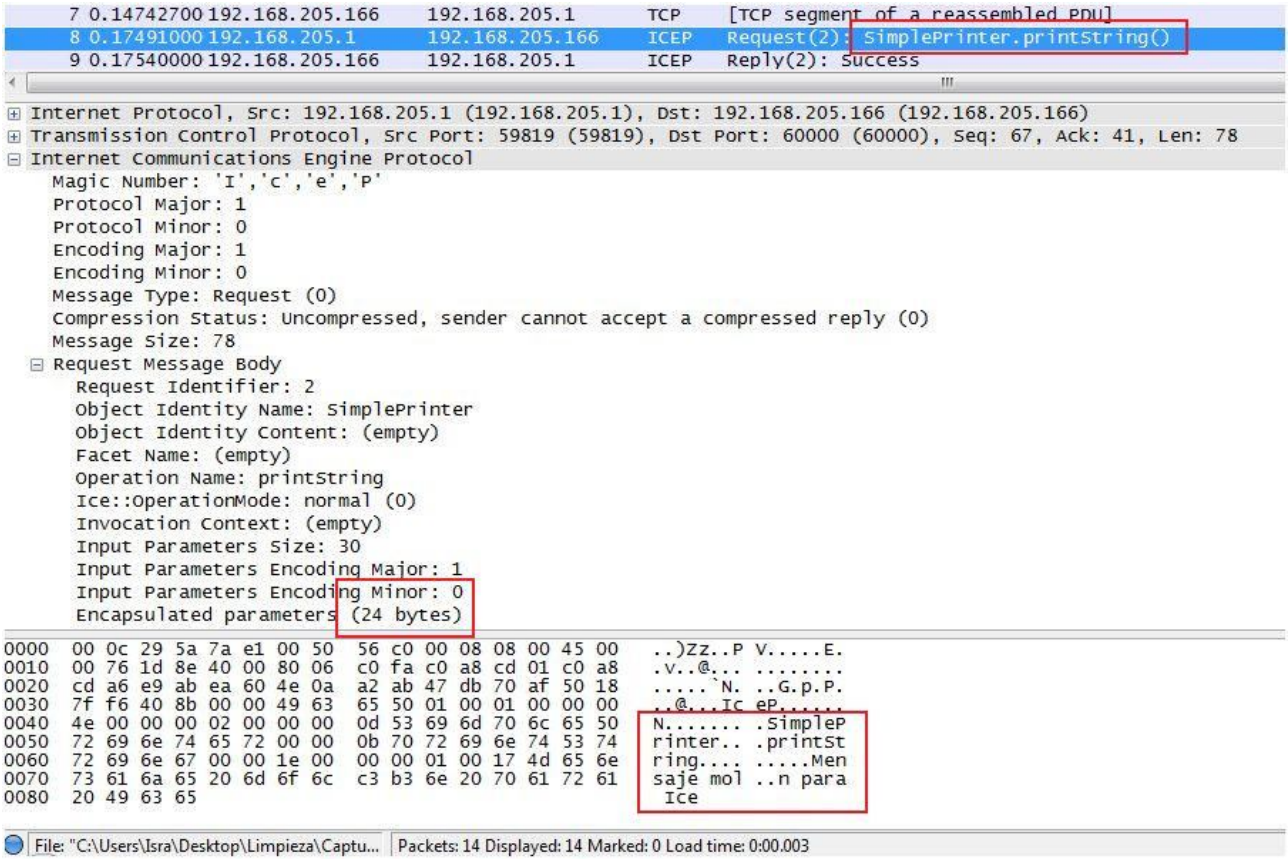


Figura 20: Captura del escenario C++ - Java

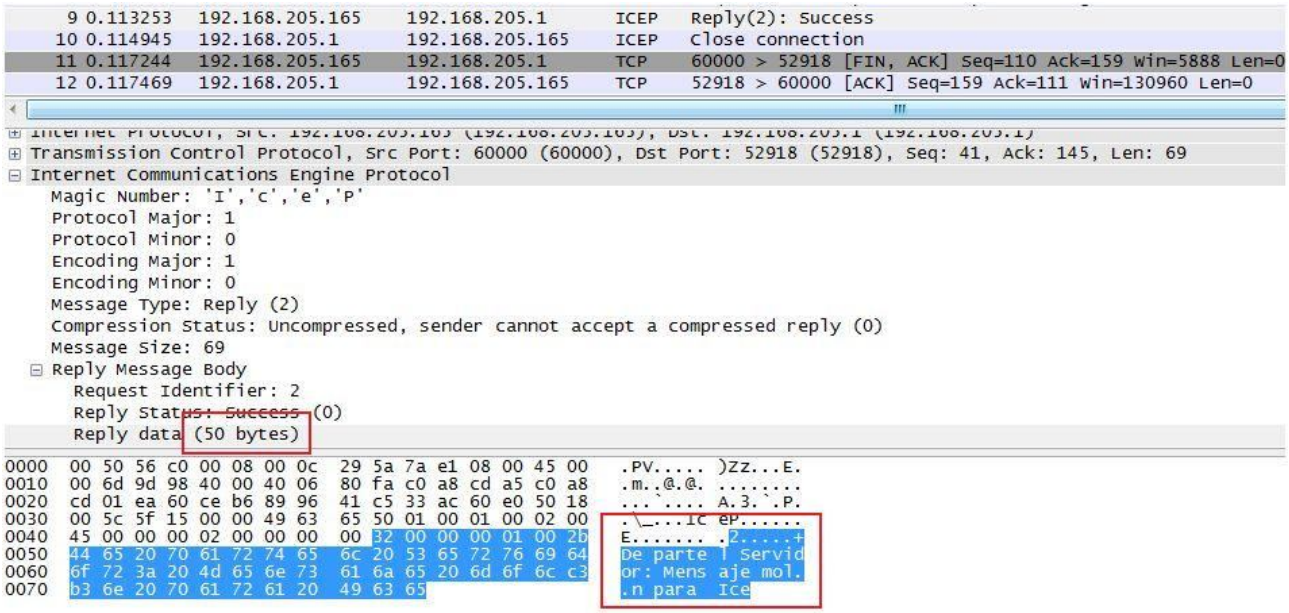


Figura 21: Captura de la respuesta obtenida para ambos escenarios

3.3 Desarrollo de una aplicación práctica

El escenario de esta prueba, puede verse de manera gráfica en la Figura 22, donde el servidor que tomará imágenes está ejecutándose en Windows y el cliente que las recibe, en Linux Ubuntu.

El propósito del desarrollo de esta aplicación, fue el de probar **ICE** bajo un entorno diferente al realizado en las pruebas de escenario. Así mismo, se probó que era factible el envío y recepción de imágenes codificadas como un *array* de bytes.

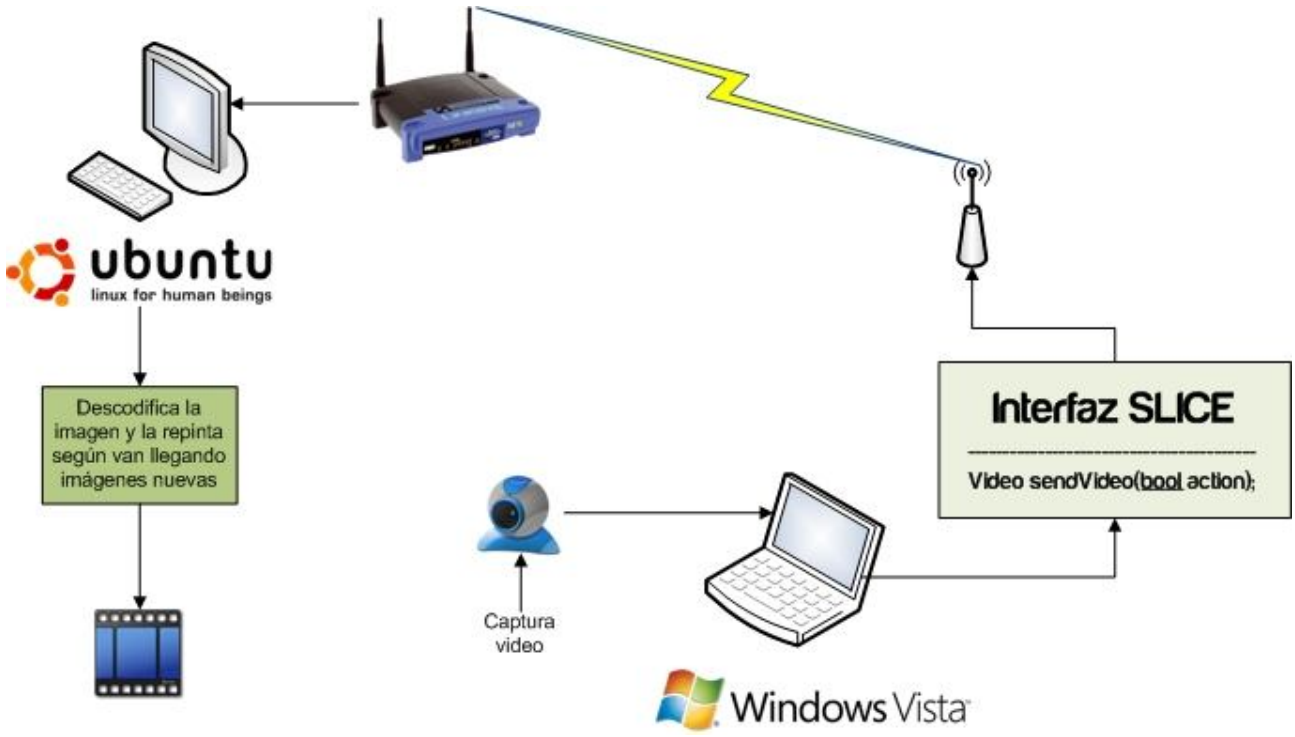


Figura 22: Arquitectura de la aplicación de video.

Como explicación resumida del funcionamiento de esta aplicación, se puede decir que el cliente realiza peticiones recursivas al servidor para obtener una imagen por cada petición. Mediante una interfaz gráfica, podrá controlar a voluntad cuando iniciar, pausar o parar ese flujo de peticiones, refrescando dicha interfaz por cada imagen que reciba, dando la sensación de estar mostrando video.

3.3.1 Uso de ICE para esta aplicación

3.3.1.1 Definición del fichero SLICE

Una de las grandes ventajas que aporta **ICE**, es la definición de un lenguaje propio el cual puede ser usado sin importar plataforma ni lenguaje (como ya se ha explicado en el capítulo 2).

En este caso, aunque se utilizó una implementación sencilla de lo que el lenguaje **SLICE** puede ofrecer, se ciñe perfectamente a lo que se busca en esta parte del proyecto.

Este archivo es el cual va a permitir la comunicación entre dos plataformas. Consta de dos partes; un módulo y un interfaz.

Creando *ImageSender,* se define el módulo en el cual va a ir el tipo de datos que se van a utilizar luego en el interfaz *Sender.*

Dado que en el lenguaje **SLICE** no define *arrays*, se ha de definir un nuevo tipo de dato el cual soporte esta estructura.

Aunque explícitamente no existen dichos *arrays*, **SLICE**, permite definir sequences, que son en esencia, *arrays*. Aquí se utilizó la siguiente línea para definir un *array* de *bytes* el cual pueda utilizarse para enviar una imagen. Se le llamó **Video** simbólicamente, ya que el cliente recibe un flujo constante de imágenes estáticas que puestas una tras otra, dan sensación de tal.

```
sequence<byte> Video;
```

A la hora de recibir/enviar flujos de *bytes*, se define un método por el cual, tras pasarle una variable booleana para empezar a enviar o no, devolverá la secuencia de bytes que se ha definido antes.

```
Video sendVideo(bool action);
```

Como se ha podido apreciar, la definición de este archivo es bastante sencilla, no obstante, se vuelve a hacer hincapié, en que cumple sobradamente el objetivo de esta implementación.

3.3.1.2 Funcionamiento de ICE en la aplicación de video

El funcionamiento de **ICE** para la aplicación de video, es una adaptación del ejemplo básico que se puede encontrar en el manual de funcionamiento de **ICE**, ya que esta implementación era uno de los primeros acercamientos que se tuvieron con **ICE** y más concretamente, con el envío de datos, que no fueran simples cadenas de texto.

El procedimiento que se utiliza para la interconexión de sistemas, es el mismo que el explicado en el apartado 3.2.2.

Aunque más adelante se explicará con todo detalle, para una lectura más sencilla se puede resumir el funcionamiento de **ICE** en unos pocos pasos:

1. El servidor y el cliente incluyen una descripción del mismo fichero **SLICE**.
2. El servidor, necesita instanciar una serie de elementos para poder compartir en red sus métodos bajo unos parámetros de conexión fijados.
3. El servidor arranca con esos parámetros y ya queda listo para usarse.
4. El cliente hace una tentativa de conexión con los mismos parámetros que el servidor.
5. Si no salta ninguna excepción, el cliente puede utilizar el servidor sin ningún problema.

3.3.2 Implementación del servidor

El servidor de esta aplicación, consiste en quedar a la espera para enviar una imagen tras otra al cliente. Funciona bajo demanda, lo que quiere decir, que si el cliente no realiza peticiones, el servidor no enviará imágenes.

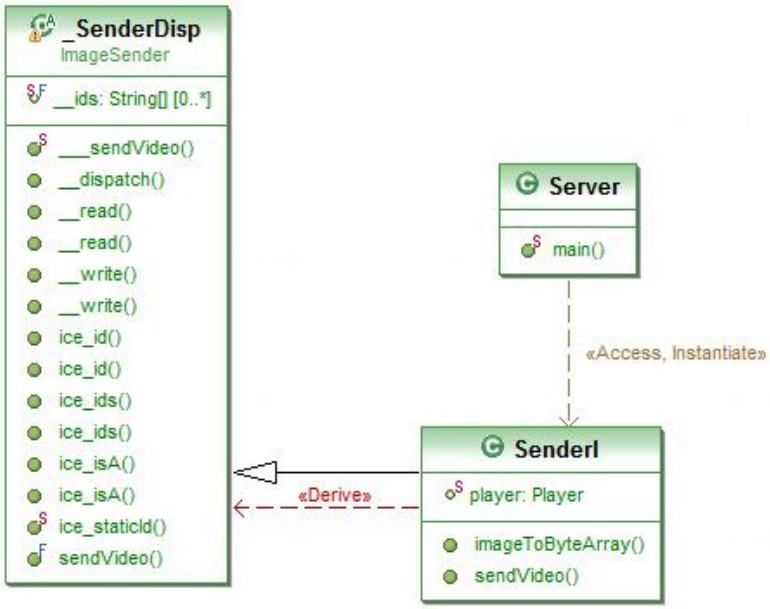


Figura 23: Diagrama de clases del servidor de "video"

Como puede verse en la Figura 23, el diagrama de clases es bastante sencillo y se compone de: Interfaz remota (*SenderI*), clase principal (*Server*) y paquete generado por el programa *slice2java* a raíz del fichero **SLICE** *(_SenderDisp*).

3.3.2.1 Interfaz remota que ofrece el servidor

Clase *SenderI*:

Tras la definición del fichero *Slice*, se tendrá que crear un interfaz por el cual implementar la funcionalidad de los métodos definidos en dicho fichero. Es decir, a la hora de declarar la clase, tendrá que heredar del módulo *ImageServer* (más concretamente, de su clase *SenderDisp*) para implementar los métodos definidos en el fichero *Slice*.

```
public byte[] sendVideo(boolean action, Ice.Current current)
```

Una vez definida, requerirá de la reescritura de los métodos definidos en su fichero Slice. En este caso solo es uno, el método *sendVideo*.

En lo referente a atributos, solo tendrá uno, un objeto de tipo *Player*, el cual utilizará (cuando esté debidamente inicializado) para acceder a la *webcam* y así poder obtener el *array* de bytes que forman una imagen.

Método **sendVideo**:

Este método tiene dos partes diferenciadas, una para la toma de imágenes, y otra para su escritura y posterior respuesta a la llamada (en forma de de *array* de *bytes*).

```
public byte[] sendVideo(boolean action, Ice.Current current)
```

Tal como se ha dicho, en el momento de su llamada se obtiene un controlador para el objeto *Player*, el cual nos permitirá obtener un fotograma en el momento. No se ha ahondado mucho en las librerías del paquete **JMF** *Java Media Framework 2.1*, ya que la extensión de éstas supera ampliamente el propósito de esta aplicación.

No obstante, cabe destacar una línea en este punto del método obviando la parte de obtener el objeto *"fgc"* el cual es abreviación de *FrameGrabbingControl, a través del objeto Player como bien se ha dicho.*

```
Buffer buf = fgc.grabFrame();
```

Tras obtenerlo sólo hay que llamar a su método *grabFrame* y se obtendrá el buffer de información el cual consiste en una imagen en un momento concreto. Tras esto, transformaremos el buffer en un fichero de imagen gracias a las librerías de **AWT** e **IO** de Java.

Y una vez se tenga formada la imagen obtenida y guardada en forma de *array* de *bytes*, se devuelve su llamada con un *return*.

Lo verdaderamente interesante de este método, es que al hacer *return* cuando se tiene la imagen formada, no se devuelve ese *array* localmente, se devuelve a través de la conexión *ICE* al que lo ha llamado.

Método **imageToByteArray**:

Este método fue creado para la transformación de un fichero en un *array* de *bytes* en forma de imagen. Crea un flujo de datos del fichero el cual se le pasa como parámetro, abre un *buffer* nuevo a raíz de ese flujo y copia todos los *bytes* de ese *buffer* (es decir, todos los disponibles, que en este caso es una imagen entera) en un *array*. Cierra los *buffers* y devuelve el objeto en cuestión.

Este método aunque parezca innecesario, deja un código más limpio y modular al no tener que hacer todos los pasos dentro de la implementación del servidor y aparte, ha sido reutilizado en todo el desarrollo del proyecto.

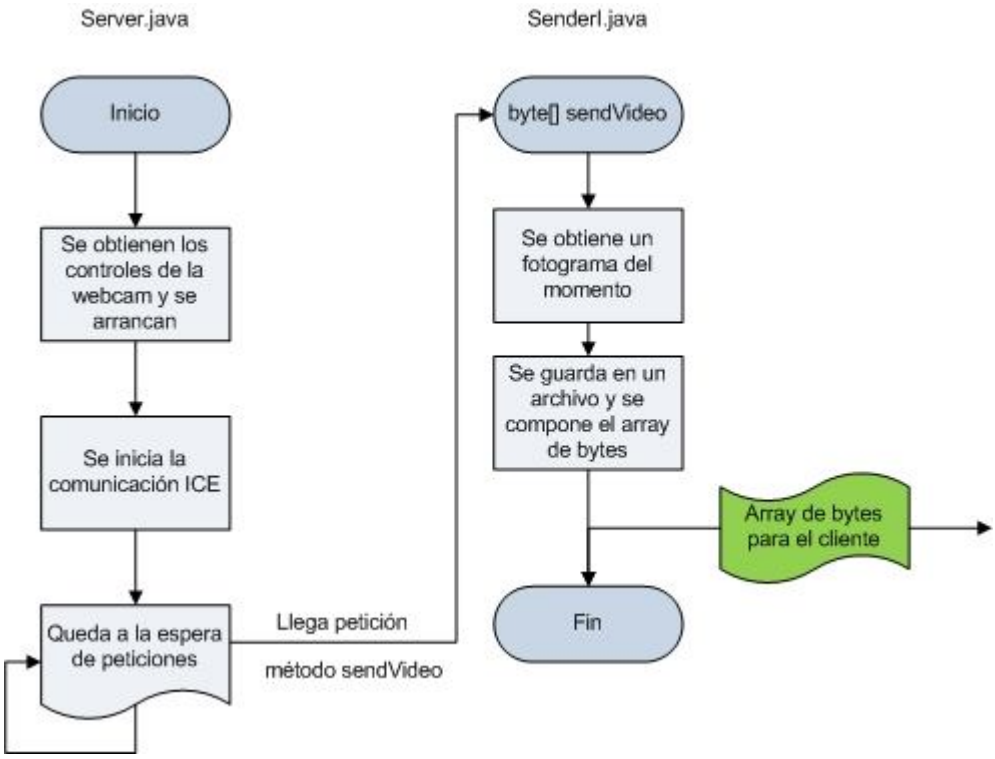


Figura 24: Diagrama de funcionamiento del servidor de imágenes

La implementación de esta parte de la conexión, se llevó a cabo en un sistema operativo *Windows Vista SP1*, plataforma la cual fue en la más sencilla a la hora de capturar imágenes de un dispositivo de video. La *webcam* utilizada fue la que viene integrada en el equipo portátil *Acer Extensa 5635ZG* por simplicidad.

Clase *Server*:

Clase encargada de mantener la conexión **ICE** y de inicializar los controladores de la *webcam*. No dispone de ningún método adicional, simplemente un método *main* el cual tendremos que ejecutar para hacer funcionar el servidor. Arranca los controles de la *webcam* y deja un segundo para su establecimiento. Tras ello inicializa la conexión **ICE** tal y como se explicó en el apartado 3.2.2 y queda a la espera de peticiones remotas.

3.3.3 Implementación del cliente

Esta aplicación gráfica se desarrolló con la idea de portarla directamente a *Android* para su uso en un terminal móvil, en cambio, se optó por desarrollar una idea más sencilla con el desarrollo de una aplicación de galería de fotos.

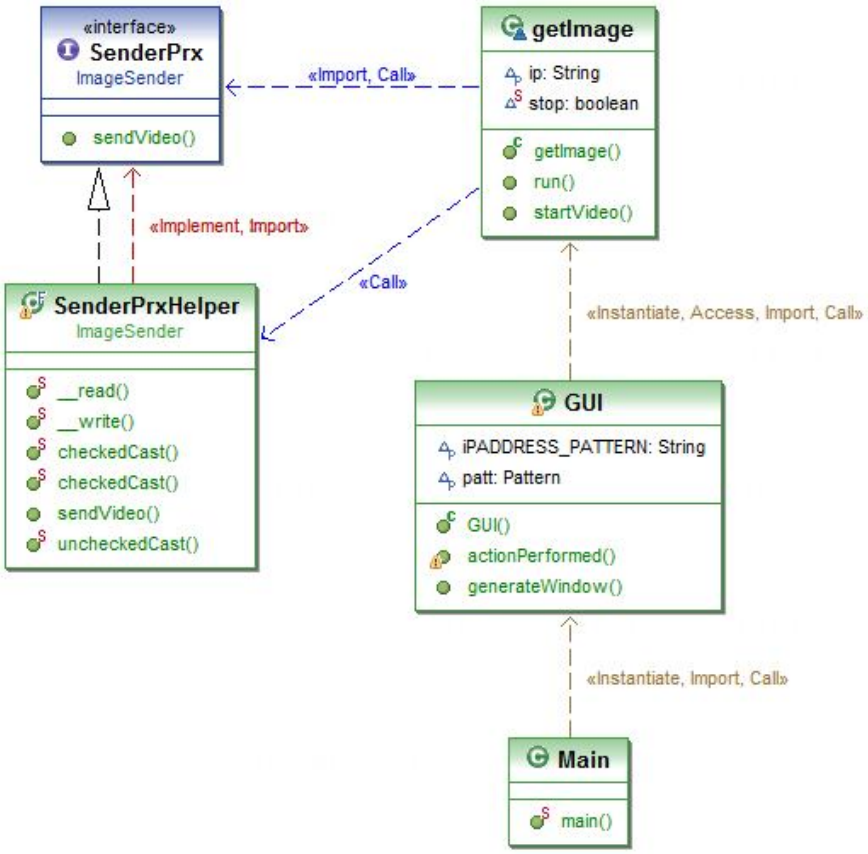


Figura 25: Diagrama de clases del cliente de la aplicación de video

Si vemos el diagrama de clases del cliente, no deja de ser una aplicación desarrollada en Java **SWING** que utiliza la tecnología **ICE** para obtener las imágenes que mostrar.

Como ya se ha explicado, este cliente deberá implementar la misma definición del fichero **SLICE** que el servidor, para que tras una conversión mediante el programa *slice2java*, se autogenere la pila de clases necesarias para poder usar dichos métodos.

Clase *GUI*:

Esta clase se encarga de toda la parte gráfica de la aplicación. Ella generará los componentes necesarios tales como botones, campos de texto, paneles, etc. para mostrarlos de una manera sencilla al usuario.

En lo referente a atributos, aparte de todos los componentes de la aplicación, cabe destacar un objeto común *getImage* el cual será inicializado en el momento de empezar a recibir video.

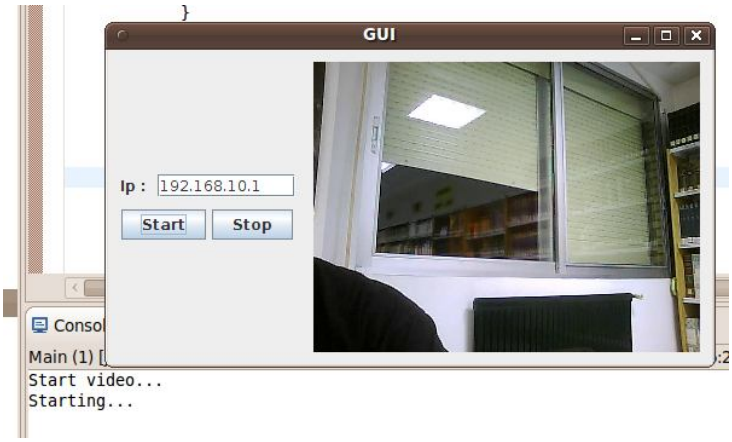


Figura 26: Aplicación de vídeo (GUI)

Método **Constructor**:

A la hora de crear un objeto **GUI**, lo primero que se hace es inicializar todos los componentes que se mostrarán al usuario. Entre ellos se encuentran paneles para colocar dichos componentes, dos botones para iniciar y parar el video, un campo de texto donde el usuario podrá escribir la **IP** del servidor el cual queremos recibir video y el área donde va a ir el propio video.

Una vez esté todo inicializado y en su sitio, se procederá a llamar al método *pack* de **ATW** y se hará visible la ventana.

Método **actionPerformed**:

Este método es el que hay que reescribir cuando se añade un escuchador de eventos a un objeto en Java. En este caso, al añadir el escuchador a los botones para detectar si se han pulsado o no, deberemos reescribirlo diferenciando qué tipo de botón ha sido, puesto que todos los eventos los lleva un mismo hilo interno de *swing*.

Lo primero que se hace en cuanto se detecta un evento, es comprobar qué hay en la caja de texto. Es decir, se añade una lógica de expresiones regulares para comprobar que lo que hay en el campo de texto, sea una **IP** de formato válido.

Continuando con la explicación, tras validar la **IP**, si el botón es el de "*start*", se procederá a crear un nuevo objeto de tipo getImage e inicializará la reproducción de video. Sin embargo, si es el botón de "*stop*", parará la reproducción actual si está en ello, o por el contrario no hará nada si no hay ningún video reproduciéndose.

Clase *getImage:*

Creada como una clase interna del archivo *GUI.java*, extiende de la clase *Thread* de Java y se encarga de hacer peticiones recursivas al servidor de imágenes. Utiliza muchos de los atributos estáticos de la clase GUI, ya que es el encargado de refrescar los paneles para que cada imagen que reciba, se muestre inmediatamente sin llegar a ser almacenada en ningún sitio, dando la sensación de vídeo.

Entre los métodos de los que se compone esta clase, al extender de la clase *Thread*, se deberá de reescribir su método *run*, el cual será llamado a la hora de pulsar el botón de "*start*" en la interfaz de usuario. Consiste en una llamada al método *startVideo* el cual se explica a continuación.

Método **startVideo**:

Llamado en el método *run* de esta clase, es el que se encarga de empezar a hacer peticiones continuas una tras de otra al servidor de imágenes y las va añadiendo al panel de imágenes, y lo refresca.

```
imgIcon = new ImageIcon(requestVideo.sendVideo(true));
                        [...]
                panelVideo.repaint();
```

Cuando es llamado, inicia la conexión con el servidor **ICE**, obtiene la primera imagen, la re-escala, la añade al panel y lo repinta. No parará de hacer esto hasta que el usuario pulse el botón de stop. Cuando esto ocurra, saldrá del bucle que le tiene encerrado, finalizará la conexión **ICE** y terminará su ejecución.

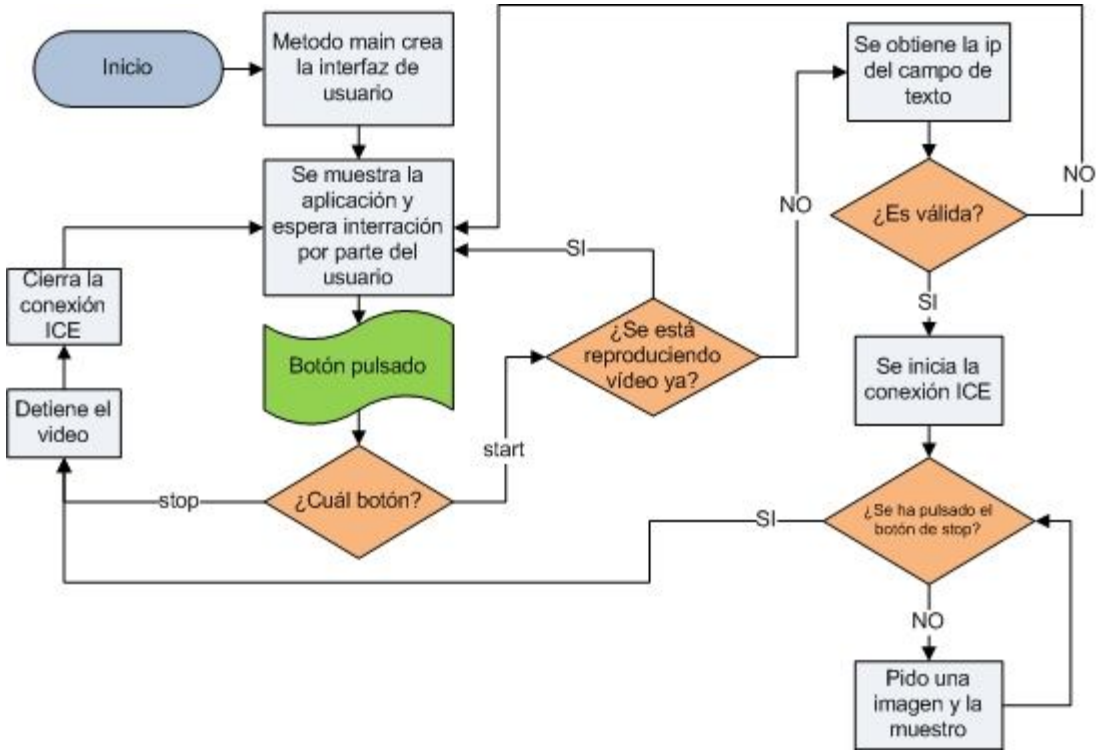


Figura 27: Diagrama de funcionamiento de la aplicación gráfica de la aplicación de vídeo

Clase *Main:*

Esta clase crea un objeto de tipo **GUI** y lo ejecuta usando la utilidad que nos provee *swing* para invocar una vez inicializado el objeto en un hilo distinto con "*invokeLater*".

3.4 Conclusiones

Aunque no se han utilizado utilidades que ofrece **ICE** tales como manejadores de hilos o de gestión de tópicos, se puede ver en manuales y tras el estudio del protocolo, que es sencillo mantener una conexión persistente.

Esta aplicación ha servido de gran ayuda para el desarrollo final de la aplicación de Android, dado que ha jugado un papel didáctico muy importante que es una primera toma de contacto con esta tecnología.

Se descartó el continuar este modelo de aplicación para Android, porque era imposible refrescar una imagen a una tasa de aproximadamente 17 **FPS**(*Frames Per Second*) la cual es aceptable para la vista de un ser humano. Es por eso por lo que se decidió crear una galería de imágenes aprovechando la experiencia en el envío de imágenes a través de **ICE**.

De la experiencia al desarrollar en el lenguaje **SLICE**, se concluye que éste es un lenguaje de definición de tipos sencillo y fácil de usar.

Además, con la ayuda de los ejemplos de los manuales y tutoriales que se pueden encontrar en la página de **ZeroC,** la implementación de un escenario básico no alberga gran complejidad.

Capítulo 4

Desarrollo de una aplicación basada en tecnología ICE y Android

En este capítulo se explica el diseño y desarrollo de una aplicación para compartir galerías de fotos mediante Android y la tecnología de *middleware* **ICE**.

Se justificarán todas las decisiones de diseño tomadas durante el proceso y mostrando sus alternativas. Se explicarán las partes más relevantes del código de la aplicación, en pos de proveer una visión más clara de la misma.

Antes de entrar en materia, cabe recalcar, que la aplicación está diseñada para fines didácticos y ha primado esto por encima de la eficiencia del mismo. Esto es debido a que se le ha dado más importancia al estudio del protocolo **ICE** interaccionando con Android.

Con esta consideración dejándola clara desde un principio, se incide nuevamente en que no serán óptimos (en algunos casos) desde el punto de vista de la eficiencia, pero demuestra claramente que la interacción Android – **ICE** es factible y en términos de *middleware* es altamente eficiente.

Figura 28: Ice + Android

El esquema que se seguirá este capítulo será el siguiente:

- **Entornos y herramientas**
- **Implementación del servidor**
  - Arquitectura del sistema y definición de la interfaz
  - Explicación del servidor
  - Explicación del cliente
  - Diagramas de funcionamiento
- **Conclusiones**

4.1 Entornos y herramientas

En el desarrollo de este proyecto, se han utilizado una gran variedad de herramientas, desde máquinas virtuales para probar la interacción entre sistemas operativos y lenguajes.

Pero eso ya está indicado en el capítulo 3 de este proyecto, en este sólo se va a hacer hincapié en el entorno *Windows – Android*, en lenguaje Java en ambas partes,

La parte de Android se desarrolló en el emulador que provee el propio **SDK** suministrado por Google. Este emulador que nos provee Google para el desarrollo en Android, cumple sobradamente sus funciones sin entrar en aspectos de eficiencia o similitud con la ejecución en un terminal físico. La parte de evaluación se hizo en dos terminales Android de distintas características, tal y como se verá en el capítulo 5

En primera instancia, se abogó por desarrollar en el **SDK** de Android *2.2*, al ser ésta la última versión disponible en el momento de la realización del proyecto. Más adelante, se tuvo en consideración que el terminal de desarrollo del cual se disponía en el laboratorio, esta versión, así pues se tuvo que bajar la versión del *API* provisto por Google a su *release* 1.6.

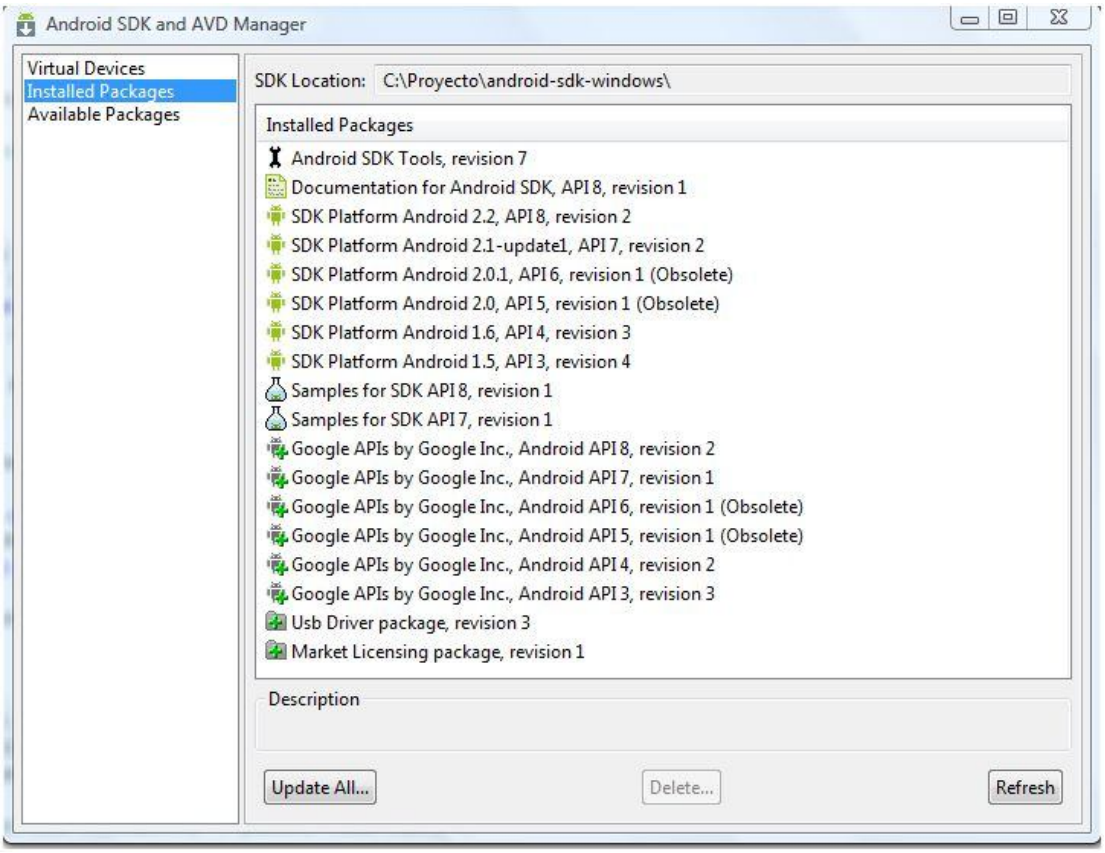


Figura 29: *SDK Manager* mostrando los *SDK* instalados para el desarrollo de este proyecto.

A la hora de crear un nuevo **AVD***(Android Virtual Device)* no se han tenido en cuenta las distintas funcionalidades que ofrece el emulador ya que no se necesita nada especial en el desarrollo del proyecto tales como acelerómetro, audio, **GPS** etc. Se hubiera podido activar la cámara del terminal (la cual viene desactivada por defecto) para poder tomar imágenes directamente desde **SharePhoot**, pero por decisiones de diseño se decidió no introducir más complejidad. Puede consultarse la lista de posibles mejoras en el capítulo 6 en el apartado de trabajos futuros.

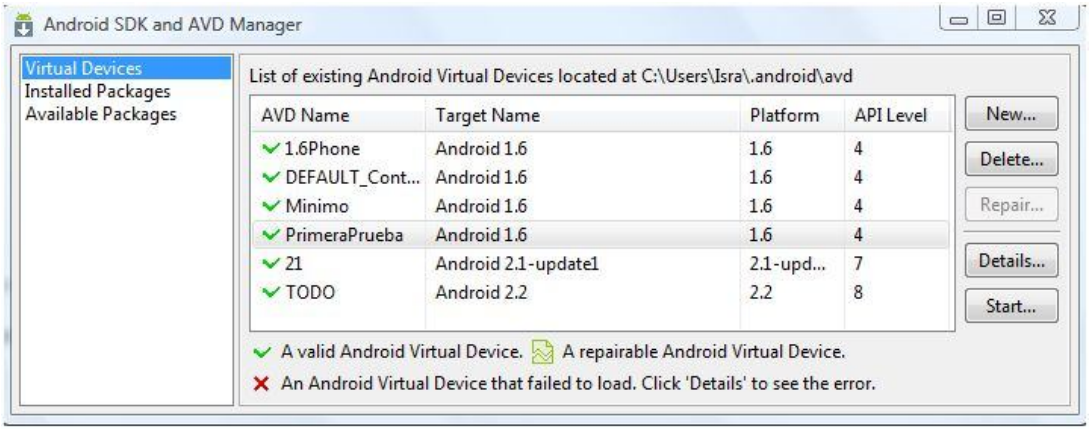


Figura 30: *AVD Manager* y los múltiples terminales usados.

Como se puede apreciar en la Figura 31, se han realizado múltiples pruebas en las cuales se ponían a prueba los distintos tipos de terminales virtuales (agregando o descartando funcionalidades) ya que no se conocía en un principio qué versión elegir. Al final, como se explica más arriba, se optó por la versión de **SDK** 1.6, haciendo prácticamente todo el desarrollo con el terminal virtual "*DEFAULT_Contodo*" sin ninguna funcionalidad extra.

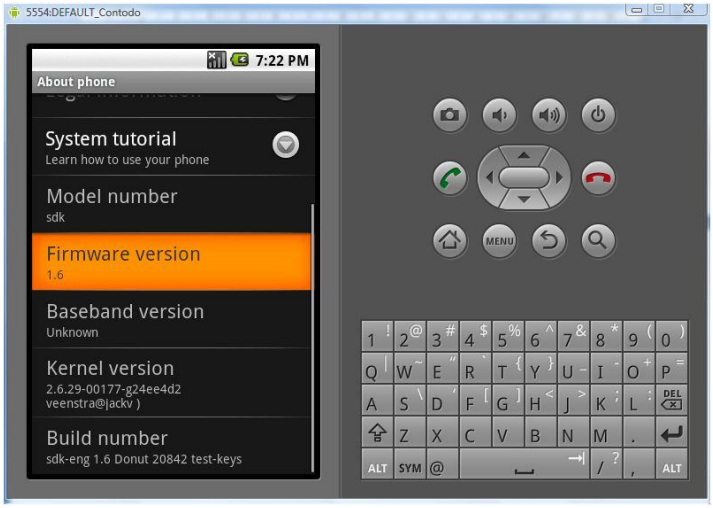


Figura 31: Especificaciones del Android *Virtual Device "DEFAULT_Contodo"* corriendo bajo *Windows*.

4.2 Implementación de SharePhoot

En este capítulo se presentará el diseño y desarrollo de la aplicación de compartición de fotos e imágenes, **SharePhoot**, programada en Java, tanto su parte de servidor como su parte de cliente. La parte del servidor se ejecuta en Windows y la parte de cliente en Android mediante una interfaz gráfica amigable y sencilla.

4.2.1 Arquitectura de la aplicación

La arquitectura de SharePhoot se puede resumir en un escenario común cliente/servidor solo que comunicándose a través de la capa de *middleware* **ICE**.

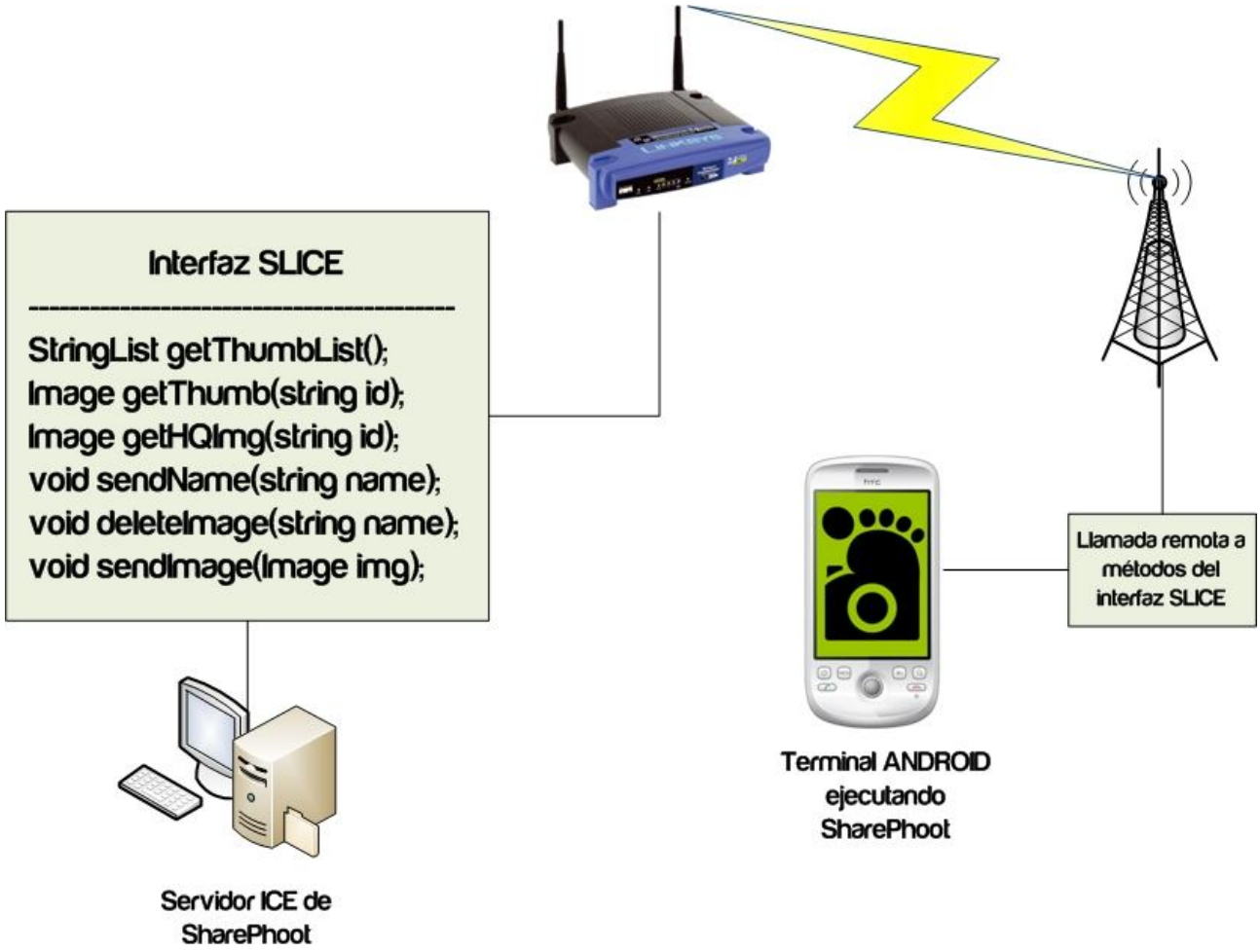


Figura 32: Escenario global de SharePhoot

Antes de empezar con la definición del fichero **SLICE**, se ha de explicar cómo funciona una conexión **ICE** en Java y por qué es necesario antes de cualquier tipo de desarrollo de código.

Cuando se empieza a desarrollar una aplicación que involucre un escenario **ICE**, es obligatorio el definir un fichero **SLICE** en el cual se indicarán las funciones, clases y atributos que compartirán las dos (o más) partes de la comunicación.

Este fichero establecerá un interfaz común a ambas plataformas para que puedan entenderse. Es por esto por lo que ambas partes de la comunicación, deben tener el mismo fichero/interfaz en sus sistemas.

Cuando se define un fichero **SLICE**, se genera toda la pila de clases dentro de su propio paquete, que se usarán posteriormente en las llamadas remotas. Así pues, cuando un cliente quiere utilizar las funciones que ofrece el servidor, sólo tendrá que tener la misma implementación del fichero **SLICE** que tenga el servidor ejecutándose.

4.2.1.1 Definición del fichero SLICE

Los métodos definidos en el archivo SharePhoot.ice, implementados por el servidor y que serán llamados por el cliente, son los mostrados en la Tabla 16.

Método	Utilidad
`StringList getThumbList()`	Devuelve la lista de direcciones de las miniaturas del servidor.
`Image getThumb(string id)`	Devuelve una miniatura de una imagen indicando el nombre.
`Image getHQImg(string id)`	Devuelve la imagen indicada a tamaño completo.
`void sendName(string name)`	Envía al servidor el nombre con el que guardar la imagen que se enviará.
`void deleteImage(string name)`	Borra tanto una miniatura como la imagen a tamaño completo del servidor.
`void sendImage(Image img)`	Envía una imagen a tamaño completo.

Tabla 16: Métodos descritos en el archivo Sharephoot.ice

Se comentarán más en profundidad los métodos descritos en el apartado 4.2.2**,** en la explicación de la interfaz remota.

4.2.2 ImageServer: El servidor de imágenes de SharePhoot.

No se entrará en grandes detalles de código del servidor, pero éste puede consultarse en el anexo **5**. Fue desarrollado acorde a las peticiones de **SharePhoot** para el manejo de imágenes de forma remota.

Este servidor llamado "**ImageServer**" se encarga de servir y almacenar las fotos de la galería. Aparte de esta tarea, se encarga de otras muchas tales como crear las miniaturas de todas las imágenes, servir esas miniaturas y fotos a resolución original bajo petición del cliente o borrar dichas imágenes (tanto la miniatura como la original) cuando el usuario lo ordene.

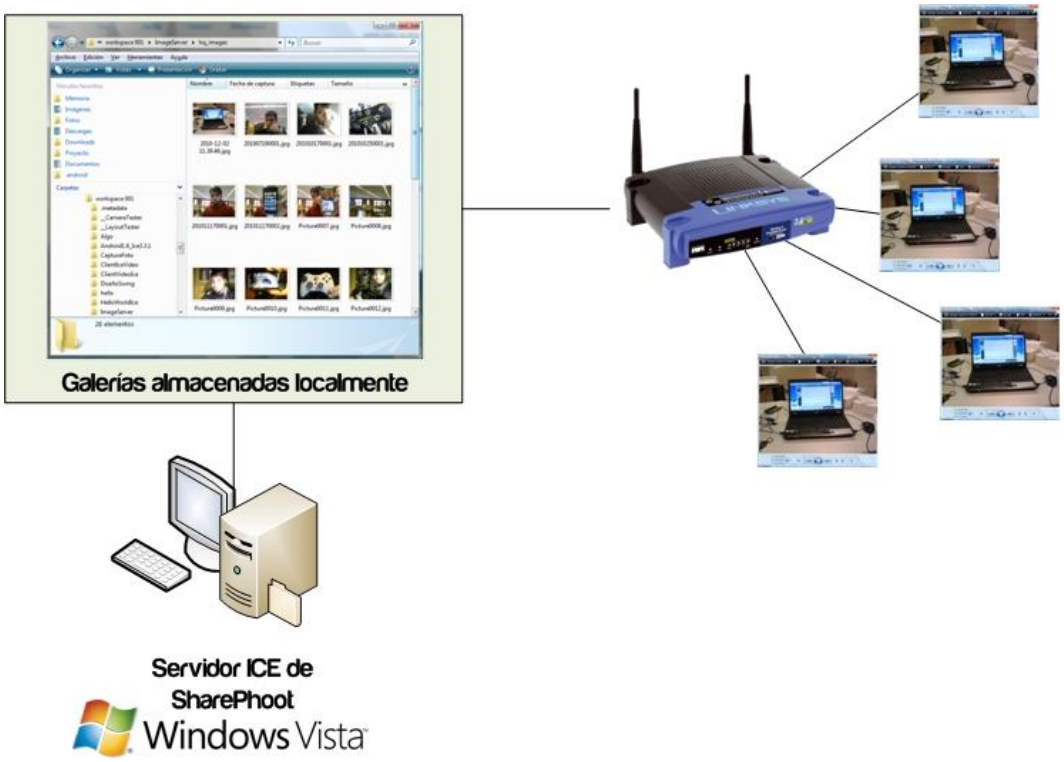


Figura 33: El servidor de imágenes de SharePhoot

Como ya es sabido, **ICE** requiere una serie de clases a raíz del fichero **SLICE** de todos los métodos y objetos que luego van a ser compartidos en red. Esta implementación corre a cuenta de la clase *ImageServerI*, de la cual se servirá el método *Main* del servidor para usar la funcionalidad asociada de los métodos compartidos. Esta jerarquía de clases se puede comprobar en la Figura 34 donde se muestran todas las clases de esta aplicación y se comenzará por explicar la interfaz remota.

La interfaz remota

Clase ImageServerI:

Una vez definido el tipo de objeto que vamos a compartir a nivel de red, en este caso *ImageServer*, se deberá construir un interfaz que extienda del objeto *_ImageServerDisp*, el cual es generado por el programa **slice2Java**. Este programa se incluye en el *plug-in* para Eclipse (puede verse su instalación en el Anexo 1). En ese objeto es donde se encuentran todas las definiciones de los métodos que se compartirán en la red.

La lista de métodos completa que tendrán que reescribirse en esta clase, pueden ser consultados en la Tabla 16 en el capítulo 4.2.1.1.

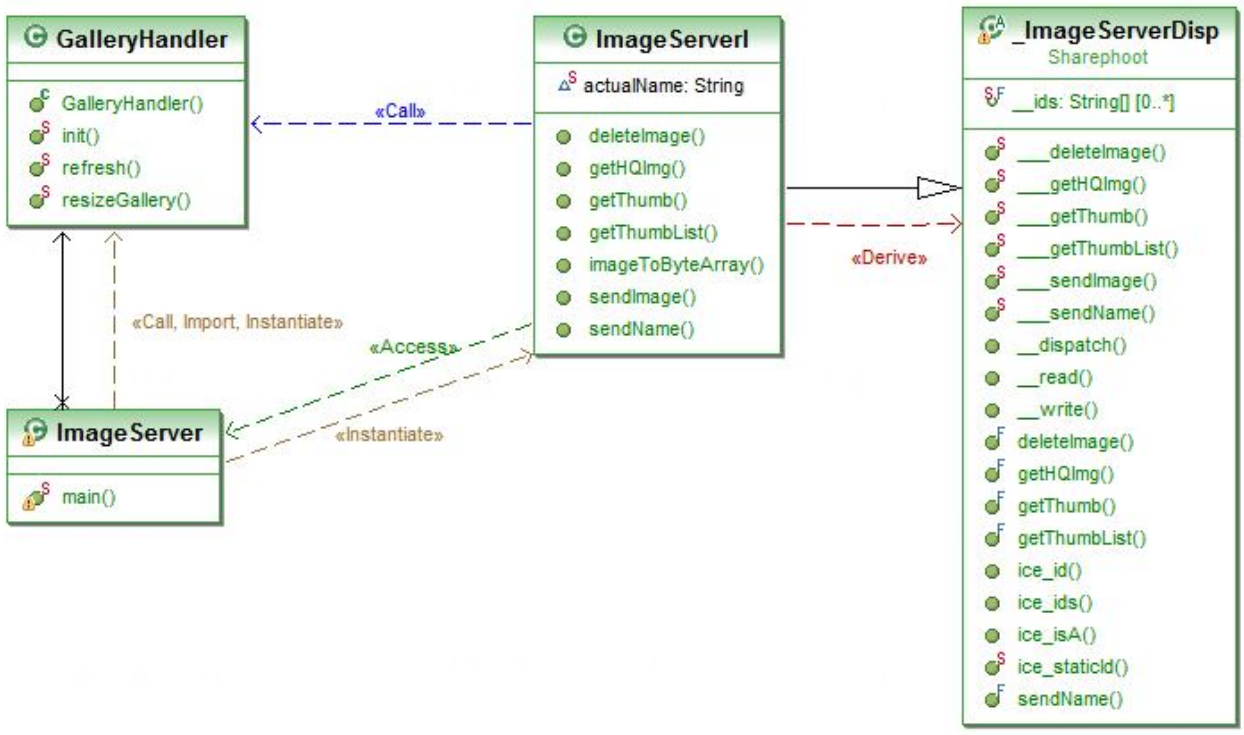


Figura 34: Diagrama de clases del servidor de imágenes de SharePhoot, ImageServer.

Método **StringList getThumbList()**:

Sirve la petición del listado de archivos. Sin él, el cliente desconocería las imágenes a las que tiene acceso y por tanto no podría pedir ninguna y su galería quedaría vacía.

El objeto definido *StringList*, es, lo que se llama en **ICE**, una secuencia de cadenas de texto, o como más se le conoce en Java, un *array* de cadenas de texto dado que en **ICE** no hay arrays propiamente dicho.

Consiste en devolver el atributo *listFiles* de la clase *GalleryHandler*. Este atributo siempre estará actualizado gracias al método `refresh()` de dicha clase *GalleryHandler*. El método y la clase se explicarán más adelante.

Método **Image getThumb(string id)**:

Este método será al que llamará **SharePhoot** a través de su visualizador de imágenes para obtener una a una las miniaturas que se encuentran almacenadas.

El cliente debe pasarle una referencia al servidor. Esa referencia se utilizará para devolver al cliente la imagen asociada a esa referencia.

El objeto que devuelve es de tipo *Image*, pero no es el tipo que se conoce en Android, es un tipo que se ha definido en **ICE** consistente en una secuencia de *bytes*. O lo que es lo mismo, el *array* de bytes del que se compone la imagen demandada.

El método `imageToByteArray(File file)` es el que se encarga de esta conversión de datos, y se explicará más adelante ya que no forma parte propiamente dicho de la implementación de la interfaz remota.

Método **Image getHQImage(string id)**:

Al igual que su homólogo `getThumb(String id)`, éste se encarga de hacer la misma función pero en el directorio de imágenes de calidad superior en vez del de las miniaturas.

Método **void sendName(string name);**

Este método se encargará de enviar el nombre bajo el cual se guardará la imagen en la galería y lo deja listo para recibir el *array* de *bytes* que formará la imagen en cuestión.

Método **void deleteImage(string name);**

Con este método es cómo se borran las imágenes del servidor, tanto su miniatura como su representación a tamaño original. Esto se consigue indicándole el de la imagen que se quiere eliminar.

Una vez tenga cerrados los flujos temporales de los ficheros los cuales ha borrado, pasará a llamar al método `refresh()` de la clase *GalleryHandler* para volver a generar miniaturas y las listas de imágenes en el servidor.

Método **void sendImage(Image img);**

En el último lugar de los métodos remotos tenemos al método que se encargará de recoger los *bytes* que formarán la imagen final y refrescará la galería para que esté actualizada.

No obstante, el servidor ha de manejar estos datos y actualizar sus galerías, cosa que es posible hacer gracias nuevamente al método `refresh()`.

Método **byte[] imageToByteArray(File file);**

Este método fue creado simplemente para una programación más estructurada puesto que esta operación no es inmediata. Dado un archivo de una imagen, éste devuelve una representación del mismo en un *array* de *bytes*.

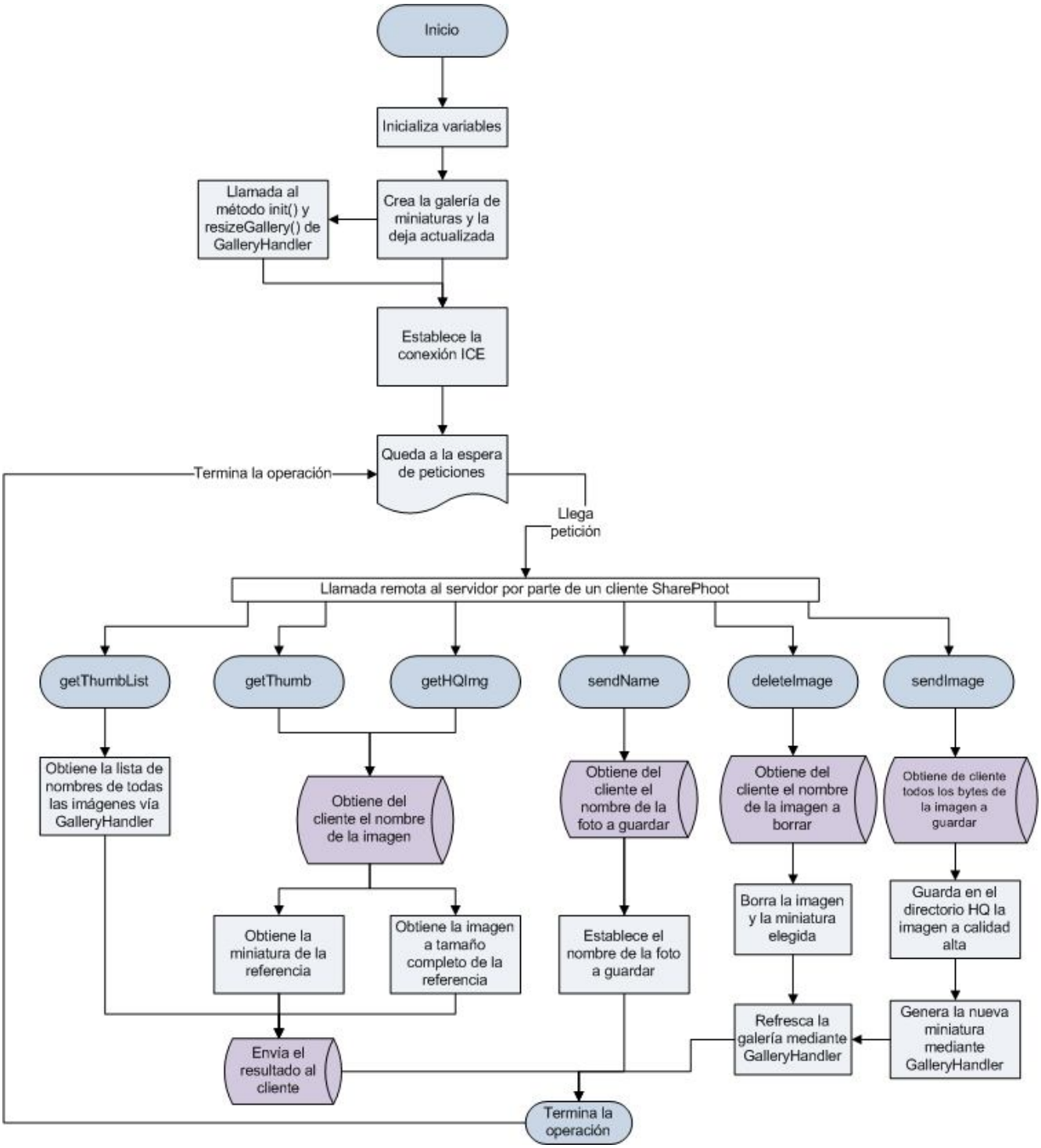


Figura 35: Diagrama de funcionamiento del servidor de imágenes de SharePhoot

El Servidor y clases auxiliares

A diferencia de su instalación, arrancar y manejar un servidor **ICE** es sencillo. Antes de poder acceder a las clases y los tipos definidos para compartir a nivel de red, se recuerda que es necesario haber indicado debidamente un módulo y un interfaz de **ICE** tal y como se explica en el apartado 3.2.2.

Este servidor compone su funcionalidad en dos clases, la interfaz que implementa toda la funcionalidad remota (*ImageServerI* la cual ya se ha explicado en el apartado 4.2.2), y la clase básica que se encarga de inicializar las galerías y encargarse de toda la conectividad **ICE** (*ImageServer*). No obstante, se entrará en detalle en la clase *GalleryHandler* puesto que es la encargada de manejar las imágenes y su funcionamiento es utilizado por el servidor para gestionar todas las peticiones del cliente, como puede verse en el diagrama de la Figura 35.

Clase GalleryHandler:

La clase *GalleryHandler*, cumple un papel muy importante en el funcionamiento del servidor de imágenes dado que ésta es la que se encarga de generar la galería que luego va a ser servida mediante **ICE**.

Aunque no sea una implementación totalmente eficiente, ya que asegura al máximo su funcionalidad, que es la de tener una galería de imágenes en un servidor local siempre actualizada ante cualquier cambio que se produzca.

Los atributos de esta clase consisten en dos *arrays* de *Strings*, *listFullPathHQ* y *listFiles*, los cuales se encargan de almacenar las direcciones a todas las imágenes en diversos formatos. De esta manera, el cliente siempre tendrá una copia exacta de las listas de ficheros que se encuentran en la galería remota.

El funcionamiento de esta clase, se puede resumir en dos métodos clave de su funcionamiento, el método `init()` y el método `resizeGallery()`, los cuales se encargan de iniciar y generar la galería de miniaturas, que luego serán servidas mediante nuestro servidor bajo **ICE**.

Método **init**:

Este método es llamado por el constructor de clase, y es el que se encarga de una vez llamado, generar las listas de imágenes que existen.

Se toma como punto de partida el directorio de las imágenes de mayor resolución, ya que nunca podrá existir una miniatura sin su imagen a tamaño completo correspondiente. Cuando llega a este punto, ejecuta el método del paquete IO de java `list()`, y con ello obtendrá el *array* de nombres de todos los ficheros de dicho directorio.

```
listFiles = directory.list();
```

Con eso se tendría cubierta la inicialización del atributo *listFiles*. Para *listFullPathHQ*, se crea un nuevo *array* del mismo tamaño y tipo que *listFiles* y se le añade la ruta completa al directorio de imágenes de tamaño original. Cuando el cliente haga una petición para las miniaturas, el método `getThumb(String id)` ya sabe dónde está su directorio y no requiere tenerlo listado.

Método **resizeGallery**:

Del método `resizeGallery()` sólo se explicará la parte más importante de éste, que es el manejo de ficheros de tipo imagen, y uno de los mayores problemas que se encontraron a la hora de utilizar este tipo de ficheros: el generar y guardar una miniatura de una imagen mucho más grande en código Java.

Este método cada vez que sea llamado, tendrá que recorrer el *array* con las rutas de todas las imágenes de la galería y generar una miniatura por cada una.

Posteriormente, la guarda en el directorio de las imágenes de calidad inferior para que puedan ser servidas al cliente.

Aunque pueda parecer trivial el generar una miniatura a partir de una imagen, esto requiere control sobre flujos y *buffers* de entrada y salida en Java. Sólo se comentarán dos líneas clave en la creación de las miniaturas.

```
img.createGraphics().drawImage(ImageIO.read(new
File(listFullPathHQ[i])).getScaledInstance(100,
       75,Image.SCALE_SMOOTH),0,0, null);
```

En esta porción de código, se muestra cómo a partir de un buffer de un fichero de tipo imagen, podemos obtener una instancia escalada de la misma, en la siguiente, cómo guardar un fichero en un directorio y que sea de tipo imagen. En este caso, **JPG**.

```
ImageIO.write(img, "JPG", new File("../ImageServer/lq_images",
                                    listFiles[i]));
```

Método **refresh**:

Aunque no hace falta entrar en más explicaciones con otros métodos de la clase *GalleryHandler*, es interesante destacar éste, que se ocupa de volver a generar las listas de archivos y las miniaturas para tener una galería siempre actualizada. Se utilizará cada vez que se añadan fotos y consiste en llamadas a los métodos `init()` y `resizeGallery()`.

Con esto queda cubierta la explicación de la clase *GalleryHandler*, la cual hace todas las gestiones con imágenes del servidor de **ICE**.

Se ha separado en este proyecto la funcionalidad externa que no tuviera nada que ver con la conectividad del servidor y para una mejor modularidad del código.

Clase ImageServer:

Esta clase es la que se encarga de mantener el servidor activo y de inicializar las galerías para que puedan ser accedidas a través de la red mediante el interfaz *ImageServerI*. Se compone solamente de un método *main* en el cual se lanzará la ejecución del hilo del servidor habiendo establecido debidamente los parámetros para que se aaccesible.

El funcionamiento de la clase es similar al explicado en el capítulo 3, apartado 3.2.2, donde se trataba el problema de lanzar un servidor en Java. El código es prácticamente el mismo con la diferencia en la línea de los parámetros de la conexión.

```
Ice.ObjectAdapter adapter =
ic.createObjectAdapterWithEndpoints("ImageServerBind", "tcp -p 60000");
```

Como bien indica la porción de código, el objeto *bind* del servidor será "*ImageServerBind*" y se utilizará con **TCP** en el puerto **60000**.

4.2.3 SharePhoot

En este apartado se explicará el funcionamiento de **SharePhoot**, la aplicación de Android que ha sido desarrollada para este proyecto. Con **SharePhoot** puedes compartir una galería de fotos de tu ordenador directamente en el terminal sin necesidad de tener almacenada ni una sola imagen en ninguna red social la cual pueda privarnos de los derechos de nuestras fotos (31)**.**

Primero se explicarán una a una las *Activities* desarrolladas para la aplicación de **SharePhoot**. Se recuerda que una *Activity* no es más que una clase de Java en Android la cual puede ser llamada por el sistema o por el programa para iniciar su ejecución.

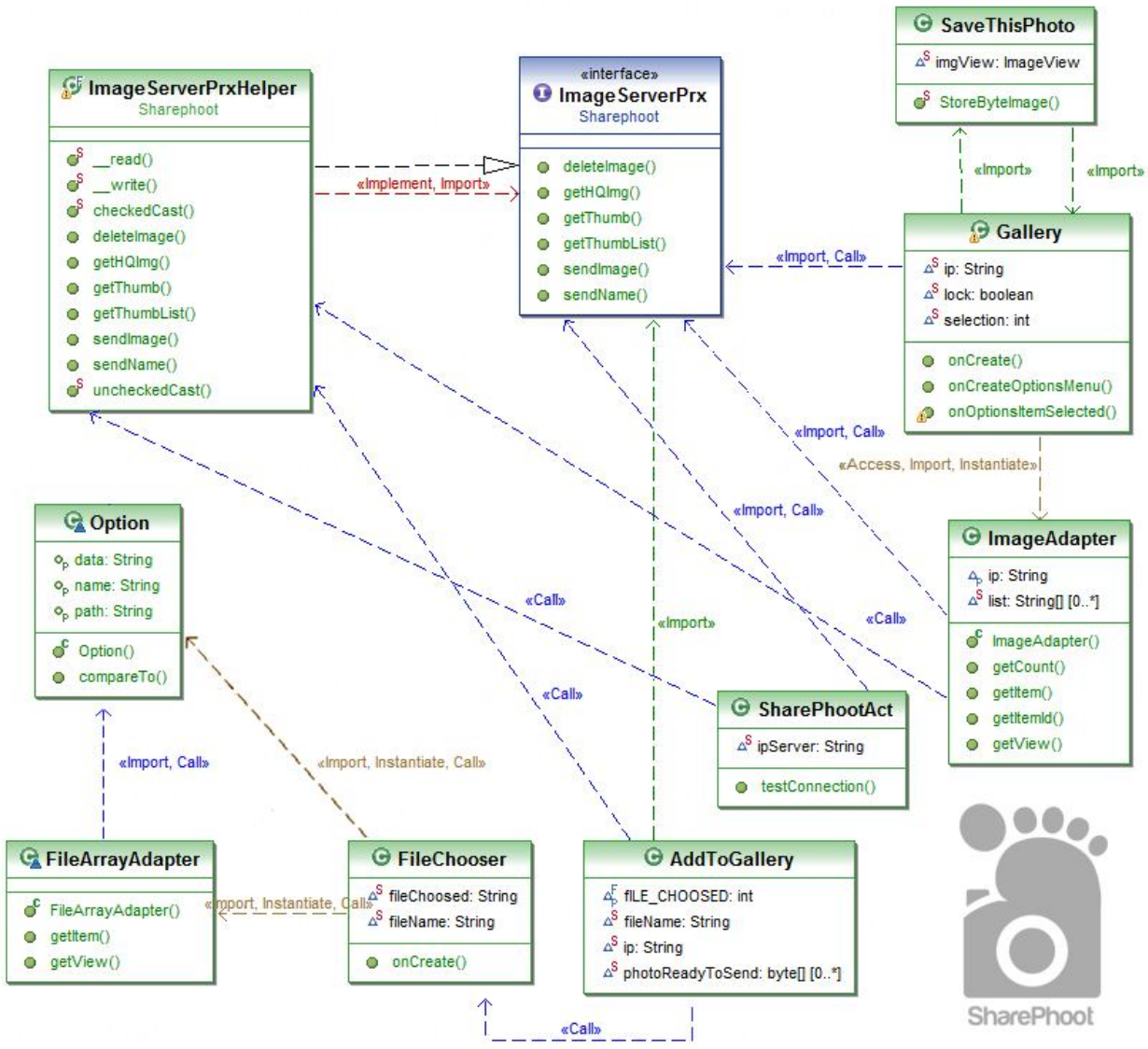


Figura 36: Diagrama de clases de SharePhoot

Activity **SharePhootAct:**

Esta *Activity*, es la más importante de toda la aplicación, puesto que cumple el papel de aplicación lanzadora (o en inglés, "*launcher*") que es la que se encarga de arrancar el programa y será la que se muestre directamente en la pantalla de nuestro terminal.

Se compone de tres estados o vistas, una imagen estática que indica el inicio de la aplicación, el menú de operaciones y la vista de ajustes de la conexión **ICE**. Estas vistas se pueden ver en detalle en el anexo 7.

Figura 37: Pantalla de presentación de SharePhoot

Como toda *Activity* de Android, deberá reescribir su método `onCreate()` el cual será llamado cuando se lance la aplicación.

Método **onCreate:**

Este es el método más importante de toda Activity, puesto que es el que se encarga de arrancar la aplicación entera (aunque no todos sus módulos).

Este método, aparte de llamar al sistema para mostrarse como aplicación, mostrará la vista principal de bienvenida de **SharePhoot** la cual se puede ver en la Figura 37, y pasados 3 segundos, cambiará a la vista del menú de operaciones o pantalla principal.

Figura 38: Pantalla principal de SharePhoot.

La pantalla principal es el punto de partida de la aplicación. Desde esta pantalla, el usuario podrá añadir sus fotos a la galería, ver su galería, cambiar los ajustes de la aplicación o salir. Esta pantalla se puede ver en la Figura 38.

Cuando la vista queda generada, se añaden los botones y los elementos necesarios. A los botones, se les añadirá un escuchador de eventos para que cuando se les seleccione o presione, la aplicación reaccione ejecutando las acciones elegidas.

Método **testConnection**:

En los casos de **Añadir**, **Galería** y dentro de **Ajustes**, en el botón "**Store**" (el cual se verá más adelante) se realizan llamadas al método `testConnection()`, método el cual se encarga de comprobar si la conexión **ICE** es viable.

Este método intenta establecer una conexión **ICE** y realizar un ping al servidor, mostrando un mensaje de error si no ha sido posible. Su código se puede ver en el anexo 6, y su explicación más detallada, dentro del capítulo 3, donde se explica cómo funciona la conectividad **ICE** dentro de *SharePhoot*.

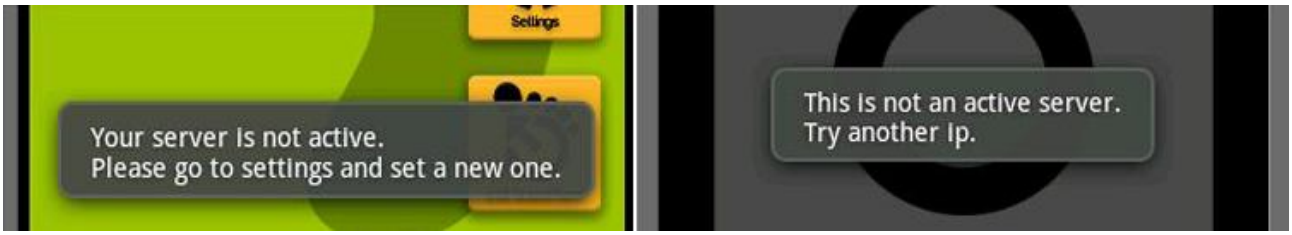


Figura 39: Mensajes de error producidos por una tentativa errónea de conexión ICE.

Función de los botones

Las acciones varían, evidentemente, dependiendo del botón pulsado o presionado. Se explicará aquí la funcionalidad de cada botón, o lo que es lo mismo, la acción que va asociada al evento que ha sido escuchado. En la Tabla 18, vemos los botones disponibles en la pantalla principal.

			
Añadir	**Galería**	**Ajustes**	**Salir**

Tabla 17: Botones de la pantalla principal de SharePhoot

Añadir (Add Phooto)**:**

Encargado de lanzar la *Activity* "*AddToGallery*" en primer plano sin finalizar la *Activity* principal. Este botón, antes de lanzar la nueva *Activity*, comprobará la conexión **ICE** mediante el método `testConnection()`. Si la prueba es exitosa, enviará como parámetro a la nueva *Activity*, la **IP** obtenida de la vista de ajustes.

Galería (Phooto Gallery)**:**

Este botón también comprobará la conexión **ICE** de la misma manera que el botón de añadir antes de lanzar la nueva *Activity* "*Gallery*". Como parámetros a la nueva *Activity*, este botón también enviará la **IP** obtenida de la vista de ajustes.

Ajustes (Settings)**:**

Esta vista es crítica a la hora de poder utilizar la aplicación **SharePhoot**. En ella se deberá introducir una **IP** de un servidor el cual esté activo. Cuando el usuario introduzca una **IP** o cualquier cadena de texto en el campo destinado a ello, lo primero que hará será comprobar el formato de dicha cadena. Si es exitoso, procederá a realizar una prueba de conexión con el servidor **ICE**. Esto se hace a través del método `initSettings()`.

Salir (Exit SharePhoot)**:**

En este botón, se ha optado por simplemente mostrar un mensaje temporal en el panel estático de Android anunciando la salida del programa y llamar al método `finish()` de la actividad principal para que se cierre.

Vista de ajustes

Aunque en el fondo la pantalla de ajustes no es más que una vista superpuesta a la principal del menú de operaciones, su funcionalidad es apartada de toda esa parte principal para no entremezclarla con la de nuevas *Activities*, ya que su uso es exclusivo de *Gallery*.

Figura 40: Vista de ajustes

Esta vista se encarga de obtener la **IP** que el usuario ha introducido en su terminal, comprobar su formato por si no fuera una dirección **IP** válida y aparte de eso, comprueba que la conexión **ICE** es posible. Los métodos que se usan tales como el ya explicado `testConnection()` y `validateIP()` el cual se explica a continuación.

El método `initSettings()` se encarga de lanzar la vista de los ajustes y de inicializar todos los botones y campos de dicha vista.

Método **validateIP**:

Uno de los problemas que se presentaron a la hora de ofrecer al usuario el poder introducir una cadena de texto, es que el cometido de dicho campo de texto, pueda usarse erróneamente y desemboque en un mal uso de la aplicación. Es por esto por lo que usando expresiones regulares, se comprobará si la cadena de texto introducida tiene el formato de una dirección **IP**.

```
[0-255].[0-255].[0-255].[0-255]
```

Activity **Gallery:**

La *Activity* de la galería es la más extensa tras la *Activity* principal *SharePhootAct*. Se compone de dos clases, *Gallery* (que será la *Activity* que lo controle todo) e *ImageAdapter* (que se encargará de generar la vista de la galería).

Figura 41: Activity Gallery

Esta parte de la aplicación, se apoya básicamente en los tutoriales de vistas que están disponibles en la página web para desarrolladores de Android (32). En este caso exactamente, se basa en una modificación de una *GridView* o dicho más comúnmente, una vista de rejilla. La forma que toma la *Activity* puede verse en la Figura 41.

Cuando la *Activity* principal (*SharePhootAct*) llama a ésta(*Gallery*), la *Activity* principal queda relegada a un segundo plano a la espera de que esa nueva *Activity* que ha lanzado, finalice. En ese momento *Gallery* se encarga de crear un objeto *ImageAdapter*, o lo que es lo mismo, crea una vista de rejilla con todas las miniaturas de la galería remota y la muestra.

Así mismo se encarga de crear y generar un menú de opciones el cual está disponible de dos maneras, presionando en la imagen que se quiere utilizar, o bien presionando la tecla menú del terminal.

Método **onCreate(Bundle savedInstanceState):**

Al igual que en *SharePhootAct*, este es el método que se encarga de arrancar la *Activity*, o dicho de otra manera, de lanzar un sub-programa sin necesidad de cerrar el anterior. En él se llama a *ImageAdapter* para crear la vista de la galería (y ésta a su vez crea la conexión **ICE** necesaria para obtener todas esas imágenes) y activa el menú de opciones para poder interaccionar con todas las fotos de la galería.

Figura 42: Menú superpuesto a la galería.

Como consecuencia de crear un menú de opciones, se tienen que reescribir dos métodos fundamentales, uno para crearlo, y otro para decidir qué hacer con la opción seleccionada.

Cuando se pulsa la tecla menú, o se llama al método `openOptionsMenu()`, *Android* internamente llama a `onCreateOptionsMenu(Menu menu)` el cual se encarga de mostrar por la parte de debajo de la pantalla el menú de opciones de la manera que se muestra en la Figura 42.

El otro método, `onOptionsItemSelected(MenuItem item)` no es más que un decisor *switch* en el cual, tras pasarle la identidad del elemento pulsado o elegido se decide qué hacer.

Menú de la galería

Las acciones que puede elegir el usuario se muestran en la Tabla 18: Iconos del menú de opciones de la galería de SharePhoot.. Estos iconos aparecerán en el menú de opciones ya que así se le ha indicado en el fichero **XML** *sharephoot_menu.xml* como se explica en el anexo 7. Se explicará en más detalle a continuación lo que hace cada una de las opciones y el código asociado a ella.

				
Guardar	**Ver**	**Refrescar**	**Eliminar**	**Cancelar**

Tabla 18: Iconos del menú de opciones de la galería de SharePhoot.

Guardar (Save)**:**

Se encarga de guardar en la tarjeta de memoria (más concretamente en la carpeta */sharephoot*) la foto que hemos seleccionado lanzando una nueva Activity que se encargará de eso mismo. La acción asociada a esta opción, es la llamada al método *savePhoto()*.

Método **savePhoto()**:

Éste método se encarga de lanzar la nueva *Activity SaveThisPhoto*. Previamente, establece dos parámetros antes de llamarla; los *bytes* de la foto en cuestión y el nombre bajo el cual se guardará en el terminal.

Una vez establezca estos parámetros, lanzará la nueva *Activity* dejando en segundo plano a *Gallery* y esperará a que termine *SaveThisPhoto*.

Ver (View Photo):

Esta acción se encarga de mostrar en una vista diferente a la actual, la imagen en grande de la miniatura que se ha seleccionado. Cuando se selecciona esta acción, se llama al método `viewPhoto()` el cual mostrará la vista descrita en el fichero *photo_full.xml* el cual puede verse en el anexo 7.

Figura 43: Resultado de la vista view photo

Método **viewPhoto()**:

En el momento en el que el usuario seleccione una foto para verla de mayor tamaño, la *Activity* principal *Gallery*, quedará en un segundo plano, mostrando la foto a pantalla completa hasta que el usuario quiera volver.

Esto lo hace mediante una petición **ICE** al servidor, indicándole a éste la referencia de la miniatura la cual está seleccionada. Entonces, el servidor no tiene más que elegir la imagen de tamaño completo de su repositorio con la misma identidad y enviarla de vuelta.

Cuando el usuario quiera volver a la galería, tendrá que presionar en la imagen para regresar. Esto se le indica mediante un mensaje cuando se abre a pantalla completa la imagen. Cuando vuelve, se tiene que llamar método `refresh()` para que funcione correctamente. Este método se explica en el siguiente punto dado que es una acción del menú en sí.

Como aclaración de por qué se tiene que llamar al método `refresh()`, se puntualiza que si sólo se mostrara la vista de la galería sin volverla a generar, ésta se habría generado sin ninguna miniatura, y por tanto, no se podría seguir con la ejecución normal del programa.

Esto es debido a la forma que tiene *GridView* de generar las cuadrículas, cada vez que es llamado, obtiene las imágenes (en nuestro caso del servidor **ICE**) y las vuelve a mostrar. Esto quiere decir que necesita un objeto *ImageAdapter* cada vez que quiera mostrarse, así que si se la lleva a un segundo plano, se tendrá que volver a generar dicho objeto, o de una manera más sencilla, se llama al método `refresh()` que hará todo esto automáticamente.

Refrescar (Refresh)**:**

Método **refresh()**:

Este método puede parecer no demasiado útil, pero es de gran ayuda para tener siempre una galería actualizada con cualquier cambio que se produzca. Su funcionamiento se basa en el lanzamiento de *Activities* como hemos hecho en otras partes del código, sólo que en este caso, la *Activity* que lanza es ella misma y la *Activity* que finaliza es ella misma también.

Por aclaración, se quiere decir que aunque se lanza una nueva *Activity*, la *Activity* que la ha lanzado, sigue en ejecución mientras la nueva está iniciándose. Así que en este caso, tras lanzar la nueva vista de la galería, la Activity que ha lanzado la nueva, finaliza y deja en primer plano a esa nueva, la cual es una nueva instancia de ésta. Si no queda claro el funcionamiento de `refresh()`, se puede consultar el esquema de la Figura 44.

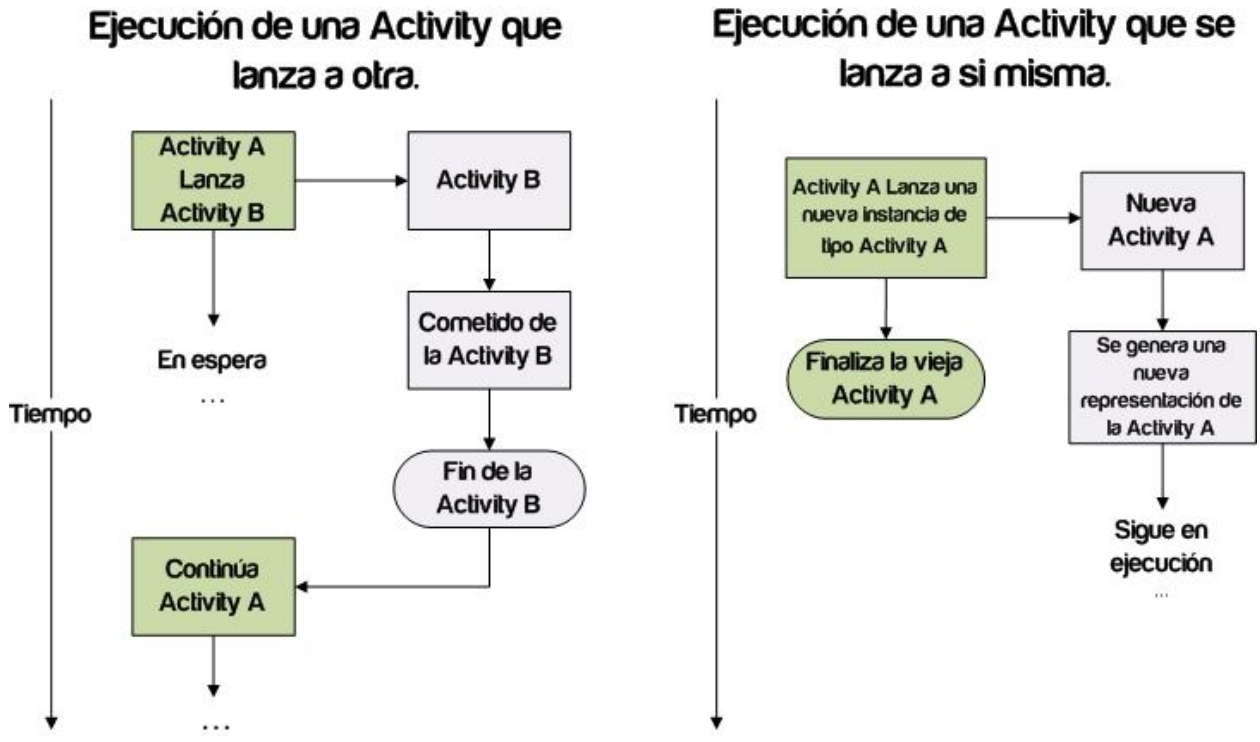


Figura 44: Esquema de cómo una Activity se refresca.

Eliminar (Delete)**:**

Esta acción de la galería consiste en hacer una petición rápida al servidor de **ICE** y pedirle que borre la imagen seleccionada. Dado que se tiene como un *ImageAdapter* objeto común y a la vez su atributo "*myServer*" accesible, se pueden hacer las peticiones directamente sin necesidad de crear más conexiones con el servidor **ICE**. Se puede consultar el código desarrollado en el anexo 6.

Cuando el usuario decida eliminar una foto, se le mostrará un mensaje de decisión y, en caso de ser afirmativo, procederá a hacer dos peticiones al servidor, una para la miniatura, y otra para la imagen a tamaño completo.

Una vez eliminadas las imágenes del servidor, se llama automáticamente al método `refresh()` para regenerar la galería con todos los cambios hechos.

Cancelar (Cancel)**:**

En cancelar sólo se llamará al método `finish()` de *Activity* para cerrar el menú y la galería y volver a la vista de operaciones principal *SharePhootAct*.

Clase **ImageAdapter:**

Esta clase es la que se encarga de generar la vista de la galería. Es también, la que se encarga de establecer la conexión con el servidor **ICE** y hacerle las peticiones.

Al ser una clase de uso común entre Activities de la galería, es muy útil que se le definan de manera estática los métodos que se comunican con el servidor, ya que así no es necesario sobrecargar la red con aperturas de conexión múltiples.

El funcionamiento es algo más complejo, pero es totalmente automático. Extiende de *BaseAdapter*, clase adaptadora de vistas que se usa en las que tienen un formato lista (o lista cuadricular, es decir, rejilla), reescribirá el método `getView()` rellenándolo con imágenes que va obteniendo del servidor. Cuando no haya más imágenes, devolverá la vista terminada del tamaño que sea y la adaptará a la vista principal de la clase *Gallery*.

Activity **SaveThisPhoto**

Esta *Activity*, es complementaria a la funcionalidad de la *Activity Gallery*. Se podría haber incluido dentro de la funcionalidad de ésta, pero se optó por un diseño más sencillo y se separó de la implementación de la galería para una mejor comprensión del código.

Figura 45: Resultado de la Activity SaveThisPhoto

Su funcionamiento se basa en, tras lanzar una petición al servidor **ICE** para obtener la imagen, guardar en el directorio *sd/sharephoot* la imagen que previamente ha tenido que seleccionar el usuario. Si el usuario no ha seleccionado ninguna imagen, esta Activity no se lanzará nunca y mostrará un mensaje de error.

En el momento en el que el usuario decida guardar una foto seleccionada, la *Activity Gallery* seguirá esos pasos y mostrará la imagen a tamaño completo con el mensaje *"Image saved."* durante 3 segundos. Cuando ese lapso de tiempo se cumpla, volverá a la galería llamando a sí misma a su método `finish()` el cual hará que se finalice y vuelva a la *Activity Galler* que estaba en un segundo plano.

Activity **AddToGallery**

Esta última Activity, se encarga de añadir imágenes desde el terminal móvil hasta la galería de *SharePhoot* que tenemos ejecutando en el servidor local. Dicho de otra manera, es cómo **SharePhoot** permite enviar imágenes desde el móvil hasta el ordenador directamente.

Su funcionamiento no dista mucho de lo explicado anteriormente en cuanto a peticiones y repuestas del servidor **ICE**, pero añade una etapa más de complejidad a la hora de manejar un terminal Android.

Figura 46: Menú principal de AddToGallery

Esta complejidad reside en la selección de un archivo del interior de la tarjeta de memoria **SD**, ya que las librerías de Android, no nos proveen de una interfaz para elegir archivos. Es por esto por lo que se desarrolló la ***Activity FileChooser***, la cual no se explicará en detalle puesto que simplemente es una Activity encargada de seleccionar un archivo, no añade nada relevante para la explicación de **SharePhoot**.

Su implementación se basó en extractos de código hallado en los foros ***</dream.in.code>*** (33).No obstante su código puede verse en el anexo **6** y el resultado de esta vista, en el anexo **7**.

Cuando el usuario ha elegido la opción ***Add To Gallery***, se le mostrará el menú principal de esta *Activity*, el que se puede ver en la Figura 46.

Llegado a ese punto, el usuario podrá hacer dos cosas, volver a atrás, o empezar el proceso para enviar una foto desde el terminal, hasta el servidor. Si el usuario elige la opción *"Pick a pic"*, se lanzará la ya conocida ***Activity FileChooser*** para seleccionar una foto de la tarjeta de memoria. *AddToGallery* quedará a la espera del resultado que envíe *FileChooser*, es decir, la dirección de la imagen a enviar.

Cuando ya ha sido elegida una imagen, se finaliza *FileChooser* de la misma manera que toda *Activity*, con el método `finish()`. Cuando vuelve a *AddToGallery*, en vez de mostrar la pantalla principal como se veía en la Figura 46, se muestra la vista de la Figura 47, dejando una previsualización de la imagen a tamaño reducido, con un botón para enviar. En el caso de que el usuario no quisiera esa fotografía, puede volver a empezar todo el proceso pulsando la tecla "*return*" del terminal.

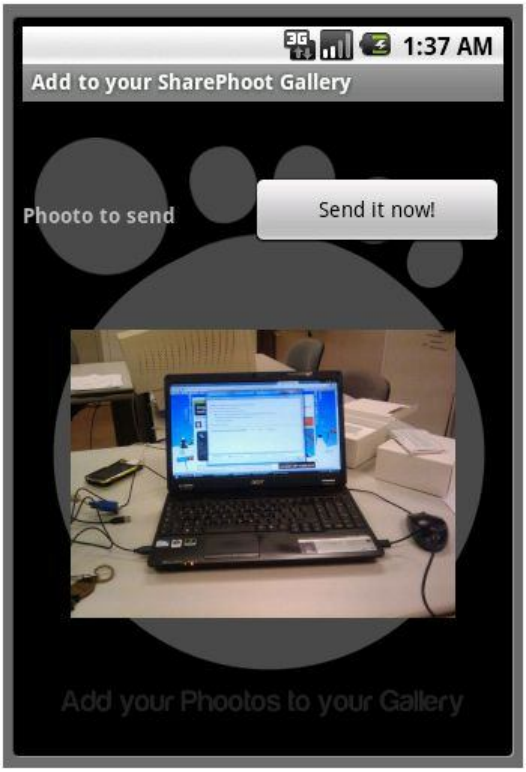


Figura 47: Pantalla previa al envío de la foto.

En el momento en el que usuario pulse *"Send it now!" AddToGallery*, lanzará una petición **ICE** al servidor para enviarle la imagen que se está mostrando. Cuando la imagen haya llegado completamente al servidor, se mostrará un mensaje de afirmación y se volverá a la pantalla de la Figura 46, donde el usuario podrá volver a realizar todo el proceso de envío, o volver al menú principal de **SharePhoot** donde podrá acceder a la galería y ver cómo la foto que ha enviado previamente, se muestra ahora en la galería compartida.

4.2.3.1 Diagramas de funcionamiento

A diferencia con los otros diagramas que han sido realizados en el proyecto, en éste no se va a proveer de un diagrama completo de todo el sistema debido a su complejidad. Se mostrarán los esquemas de funcionamiento por cada Activity referenciando siempre quién lo ha llamado y en qué condiciones.

En el caso de las *Activities* *FileChooser* o *SaveThisPhoto*, no se ha requerido el crear un diagrama aparte ya que la explicación de su funcionamiento está incluida en otros diagramas.

Activity **AddToGallery**

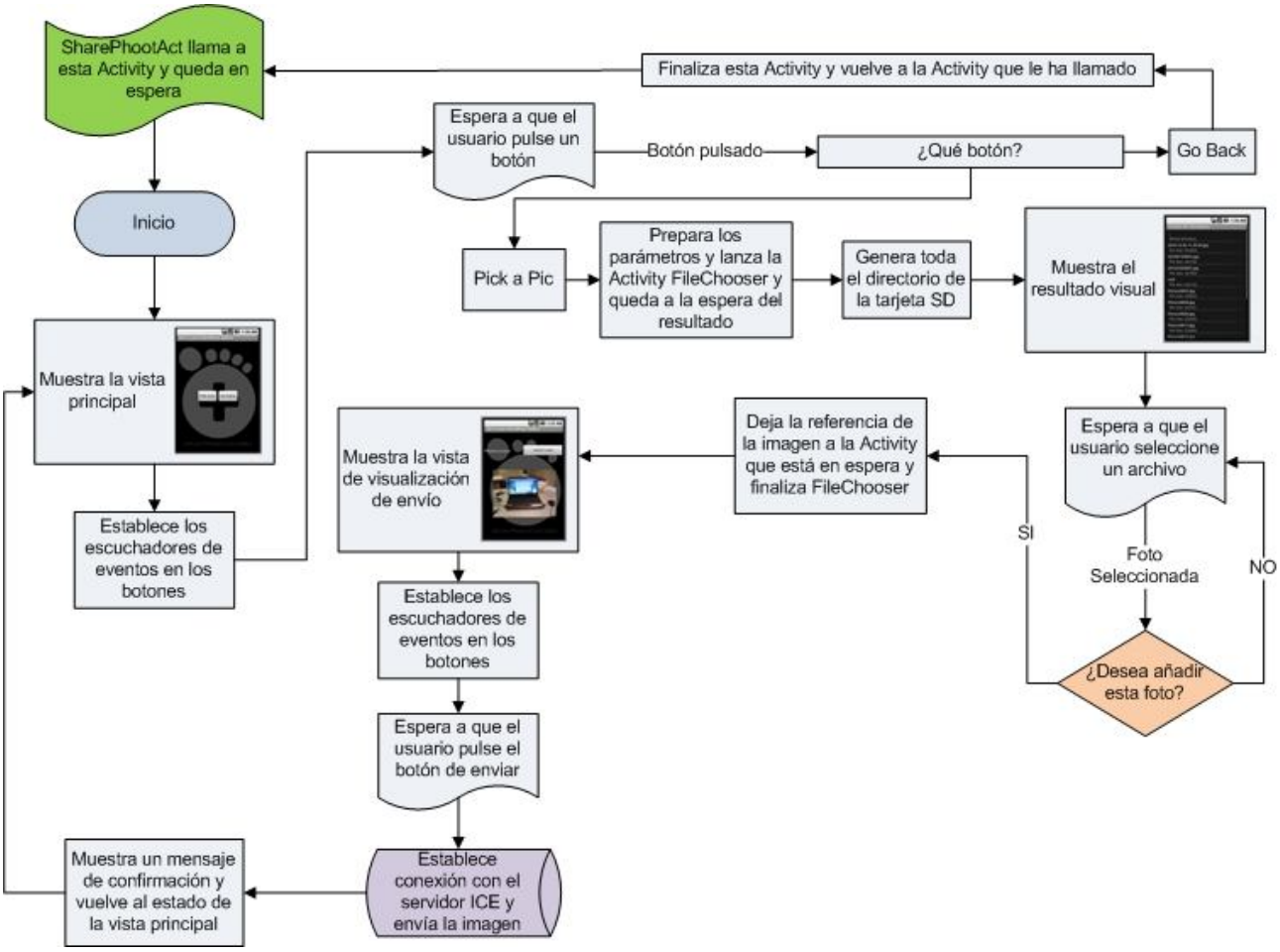


Figura 48: Diagrama de funcionamiento de la Activity encargada de añadir imágenes a la galería de SharePhoot

Activity **SharePhootAct**

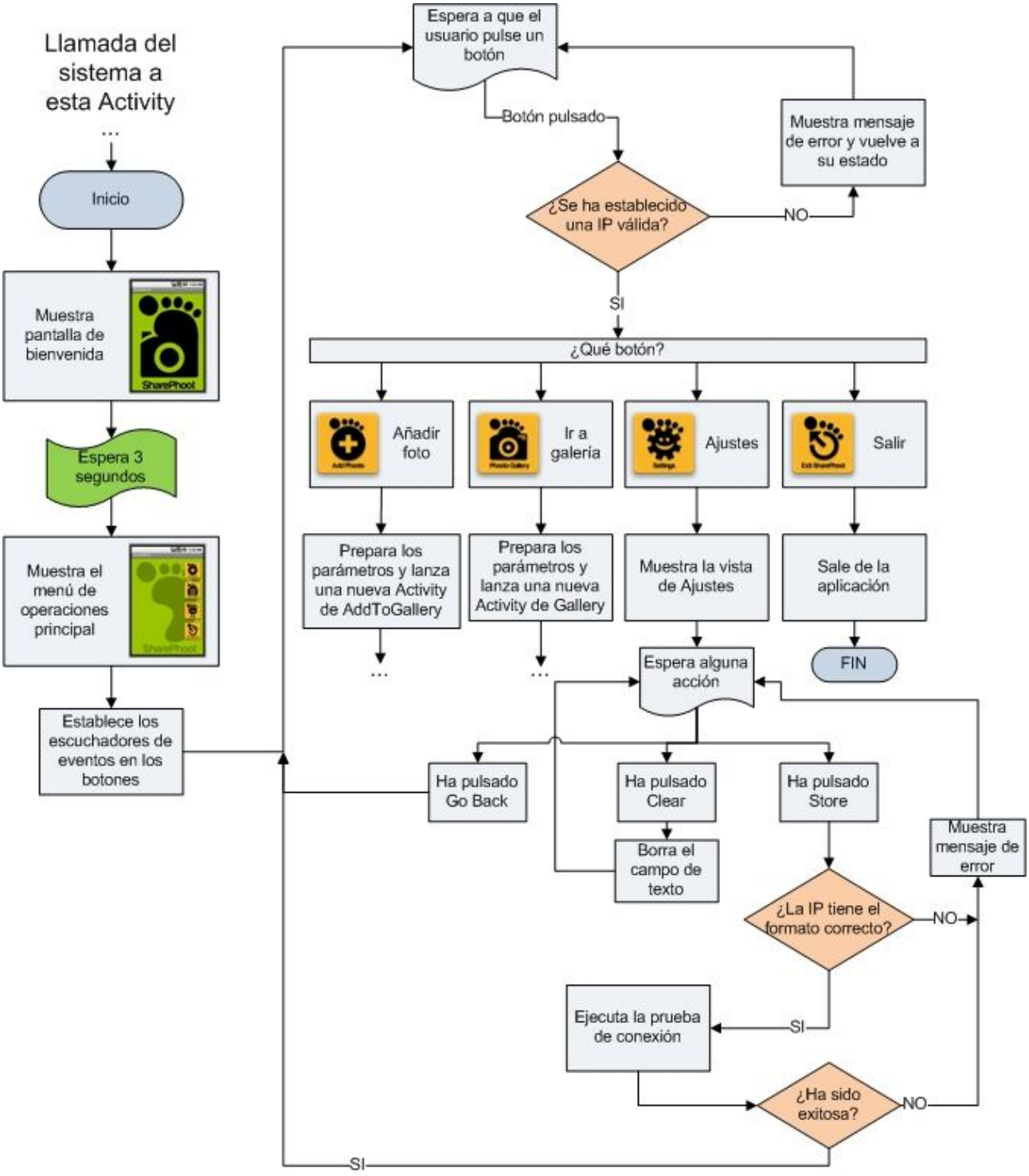


Figura 49: Diagrama de funcionamiento de la pantalla principal de SharePhoot

Activity **Gallery y SaveThisPhoto**

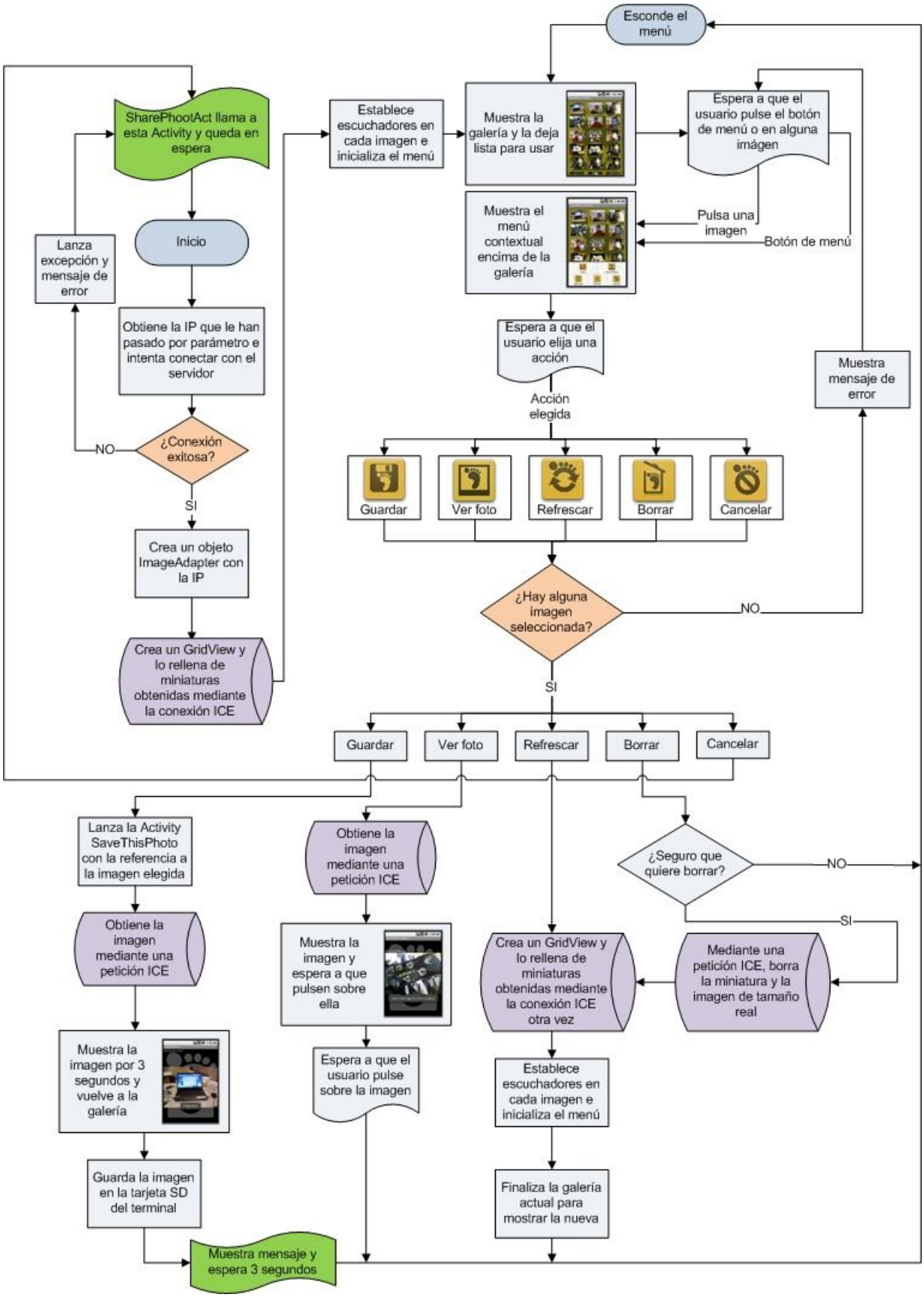


Figura 50: Diagrama de funcionamiento de la galería SharePhoot

4.3 Conclusiones

Los comienzos con Android pueden ser difíciles de entender sobre todo a la hora de programar las aplicaciones gráficas si se está muy familiarizado con el entorno **SWING** de Java, ya que este cambia radicalmente. El entorno de **SWING** no es más que código Java el cual se estructura para generar toda la parte gráfica. En Android la parte gráfica se basa en esquemas **XML** donde se describen los elementos a mostrar. Así que en cuanto se obtiene cierta habilidad con dichos esquemas **XML**, simplifica mucho la tarea que se puede desarrollar en **SWING** en otra plataforma.

Cuando se empezó el desarrollo de **SharePhoot**, en una primera instancia se intentó continuar el desarrollo de la aplicación de video, pero al final, viendo que la parte gráfica de ambos cambiaba radicalmente, se optó por crear una galería de fotos compartida directamente con un ordenador desde Android.

Las posibilidades de Android son enormes, y como conclusión se puede sacar, que cuando se empieza a programar con Android, no se tiene una visión real de lo que se puede llegar a hacer con esta tecnología.

En el comienzo del desarrollo de Android, no se comprendía entre Activity y vista. Las vistas podrían considerarse como “paneles” en los cuales mostrar los resultados de la ejecución del código, mientras que una Activity es más un programa independiente con su propio ciclo de vida. Es decir, una vista se puede mostrar u ocultar según queramos, pero siempre estará ahí, mientras que si se finaliza una Activity, ésta desaparece completamente.

Es por esto por lo que en el código de **SharePhoot**, aparecen mezcladas las vistas con las Activities, y se ha llegado a la conclusión, que para una programación más modular y más eficiente (ya que se tiene mucho más control del estado de ejecución de la aplicación) es implementar cada funcionalidad en una Activity separada.

Con el desarrollo de **SharePhoot** se ha demostrado que es factible el uso de la tecnología **ICE** sobre Android.

No obstante, se concluye que la tecnología **ICE** tiene un gran inconveniente: Es muy costosa y compleja la instalación del sistema, siendo un gran problema para un usuario medio el cual no está acostumbrado a estos niveles de manejo.

Se concluye entonces que la instalación del sistema **ICE** puede generar problemas de versiones ya que entre ellas, muchas no son compatibles.

Esto es causa de haber experimentado muchos problemas con el cambio de versiones, ya que debido a problemas de implementación interna, **ICE 3.4.1** no es compatible con la release **0.1.1** de **ICE** for Android, sólo es compatible con la versión anterior, la **3.3.1**. Se desconoce si en un futuro se solventarán estos problemas de compatibilidad de versiones.

Capítulo 5

Evaluación

En este capítulo se pone a prueba la aplicación SharePhoot en distintas versiones de Android y en distintos terminales físicos. También se realizó una prueba de rendimiento consistente en envíos de distintos tamaños de imágenes.

5.1 Emuladores de Android

Al inicio del proyecto, no se requirió de terminales físicos dado que Google en su SDK nos proporciona, como ya se ha visto, de un entorno de pruebas completo.

Esto incluye a su emulador virtual de terminales Android, y sobre éste, se han realizado pruebas de funcionamiento para las versiones de Android 1.6 y 2.2.

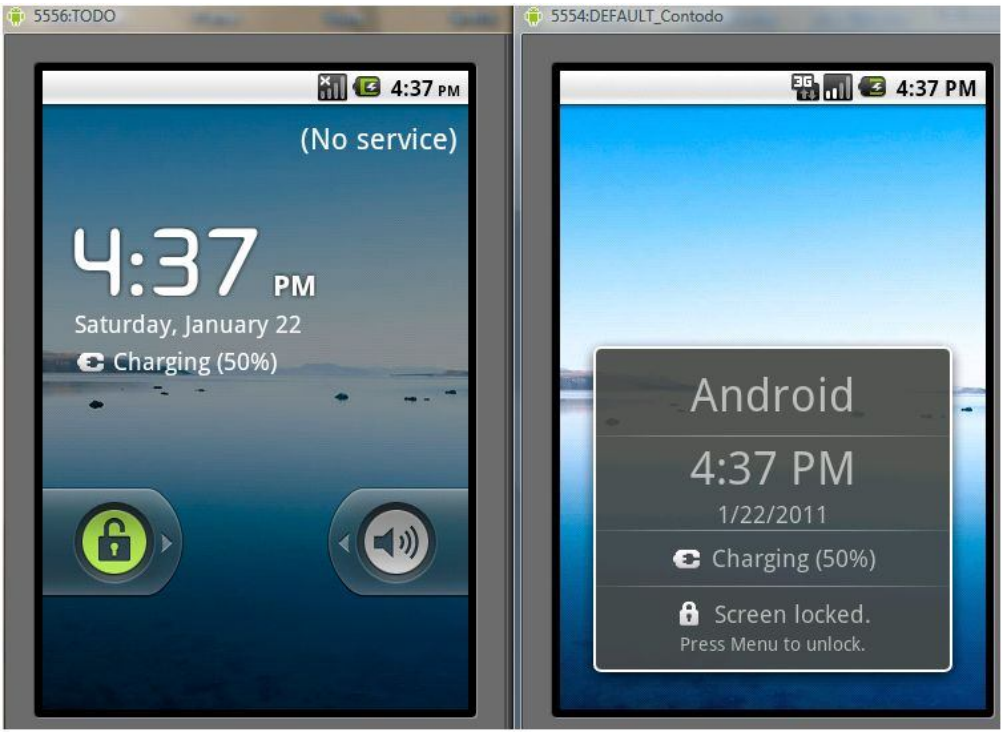


Figura 51: Diferencias entre los terminales Android 2.2 y 1.6 bloqueados

Se disponía del terminal **Google's Developers Phone 1** en los laboratorios para pruebas. Éste, como ya se verá en el apartado 5.2.1, como sólo soportaba la versión 1.6, se decidió desarrollar bajo esa versión.

No obstante, en sus inicios se desarrolló en la última versión disponible en el mercado. Es decir, con la versión 2.2 y cuando se decidió que tendría cabida en un terminal físico, se hicieron pruebas en ambas versiones, centrándose más en la versión 1.6.

Esto incluye, probar la funcionalidad de **ICE** para Android de **SharePhoot** para las versiones 1.6 y 2.2 de Android. Dado que se ha implementado la aplicación para que sea compatible en las dos plataformas, las pruebas realizadas en Android 2.2 o Android 1.6 son indistintas y no se aprecian cambios.

Podemos ver, que Android 2.2 y Android 1.6 se diferencian notablemente en apariencia si observamos las Figura 51 y Figura 52 realizadas en los emuladores virtuales.

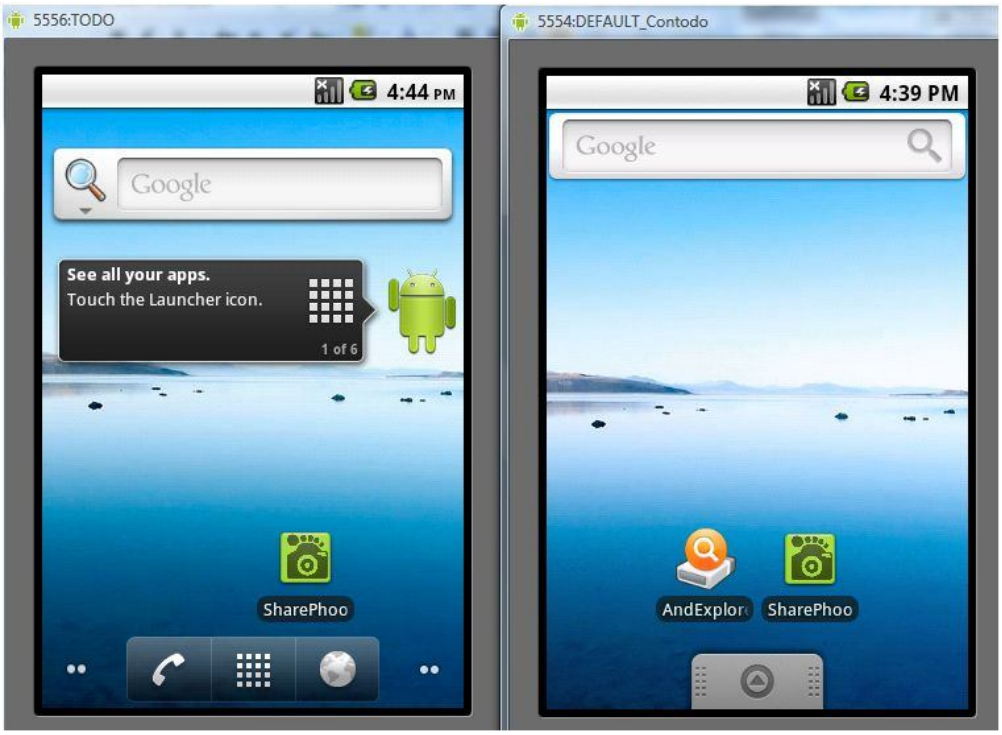


Figura 52:Diferencias entre el escritorio de los terminales Android 2.2 y 1.6

Como no hay forma de saber en qué plataforma se están desarrollando las pruebas, se ha decidido no incluir ninguna captura realizada en el emulador. Aunque a lo largo de la memoria de este proyecto, cada imagen que muestra la apariencia de alguna *Activity*, prueba de funcionamiento en el emulador.

Puede encontrar una lista completa de figuras en el índice de figuras de las primeras páginas.

5.2 Terminales utilizados

5.2.1 Google's Developers Phone 1 (Android 1.6)

Las pruebas con el terminal de **Android, Google's Developers Phone 1** sobre la versión 1.6, fueron realizadas en las instalaciones de la Universidad.

Cabe destacar, que gracias al emulador, en la primera prueba sólo se obtuvieron los siguientes fallos:

- No se implementó control de giro y esto provocaba la deformación las imágenes pero siendo totalmente funcional.
- No se implementó control de apertura de teclado y reiniciaba la aplicación.

Estos fallos se solventaron rápidamente añadiendo al fichero `Android_manifest.xml`, un par de líneas para cada *Activity* evitando la alteración visual de la aplicación.

```
<activity [...]
                android:screenOrientation="portrait"
                android:configChanges="orientation|keyboardHidden">
```

Con estos dos atributos en la declaración de la Activity, se asegura que la aplicación, siempre va a estar orientada verticalmente, y que cuando se despliegue el teclado físico, no cambie su apariencia.

La parte del teclado puede resultar un tanto ilógica, que el abrir el teclado del terminal, pueda alterarse visualmente la aplicación. Se puede comprobar en la Figura 56, que cuando se desliza el teclado, se está obligando al usuario a girar el teléfono para poder escribir con él. Y por tanto, aunque ya se haya obligado a la *Activity* a sólo mostrarse en posición vertical, cuando se cierra el teclado, lanza un evento interno y corta el flujo normal de ejecución de la aplicación.

Figura 53: Panel de aplicaciones de Android

Figura 54: Pantalla de carga de SharePhoot

Figura 55: Menú de actividades de SharePhoot

Figura 56: Pantalla de ajustes mostrando cómo se evita la deformación de la vista al desplegar el teclado

Figura 57: Misma función que la figura anterior pero con el menú de actividades

Figura 58: Pantalla principal del proceso para añadir a la galería remota

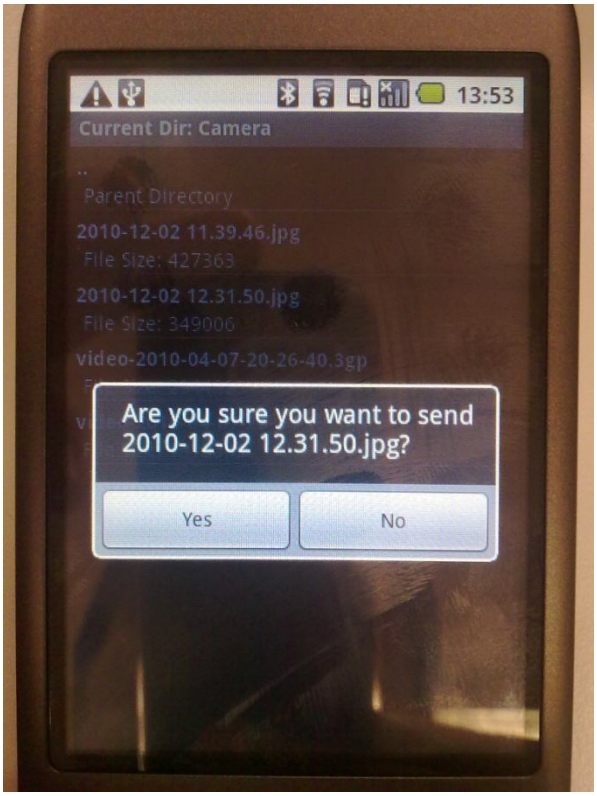


Figura 59: Pregunta de confirmación de la Activity FileChooser sobre una imagen

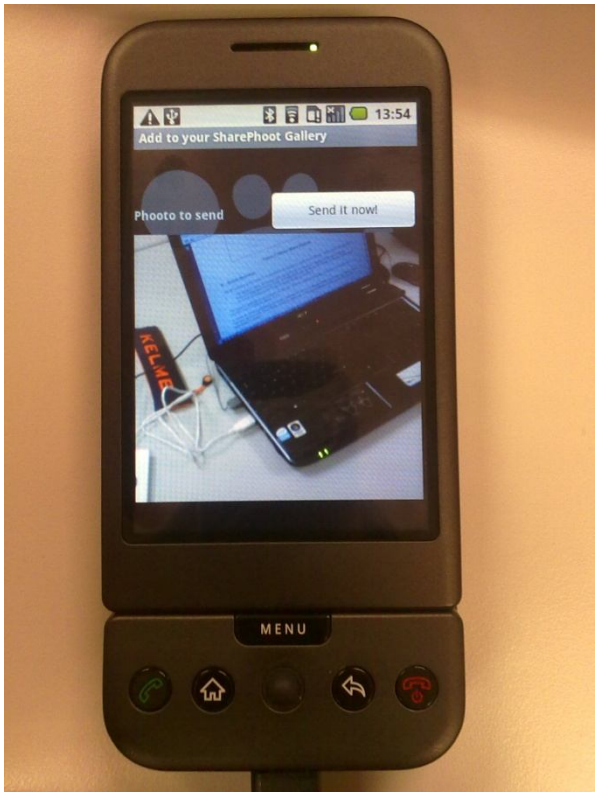


Figura 60: Pantalla previa al envío de la imagen a la galería remota

Figura 61: Galería de fotos con el menú superpuesto

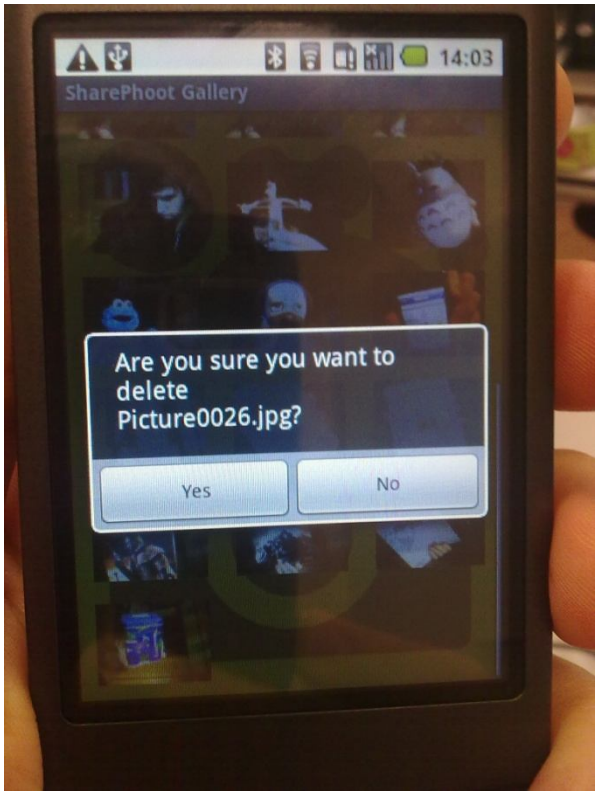


Figura 62: Pregunta de confirmación por la eliminación de una imagen

Ahora, podemos ver en las siguientes fotografías, cómo **SharePhoot** se comunica con el usuario mediante mensajes. Estos son algunos con los que cuenta.

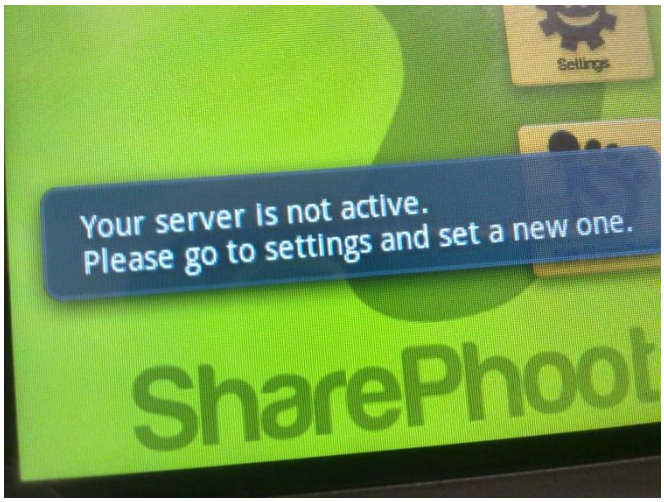


Figura 63: Tentativa fallida de conexión con el servidor ICE

Figura 64: Tentativa satisfactoria al responder el servidor ICE

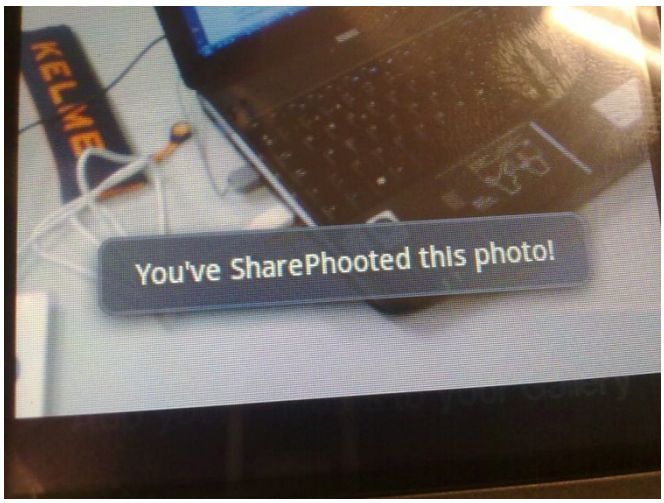


Figura 65: Mensaje de confirmación al enviar una imagen del dispositivo móvil a la galería remota

5.2.2 HTC Magic (Android 2.2)

Se muestran aquí, las fotografías del terminal **HTC** Magic con la versión 2.2 haciendo funcionar **SharePhoot**.

Figura 66: Versión del sistema operativo del terminal HTC

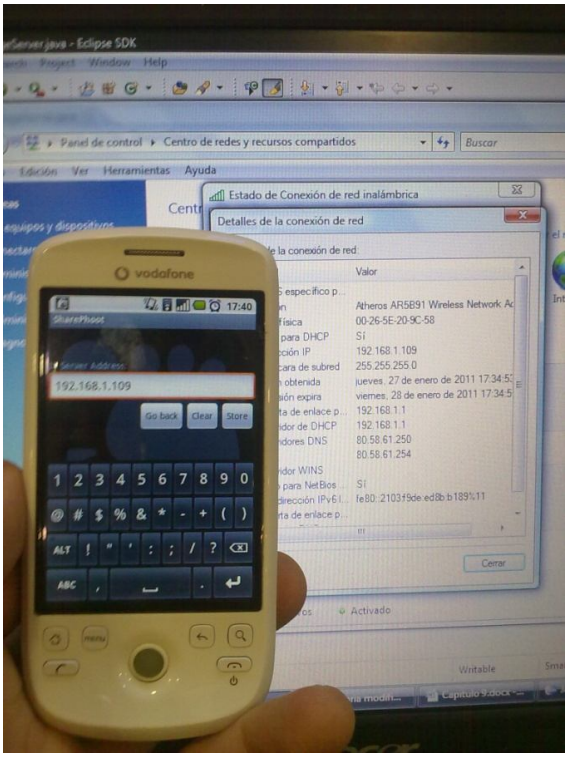


Figura 67:Pantalla de ajustes mostrando la IP en la que está ejecutándose el servidor

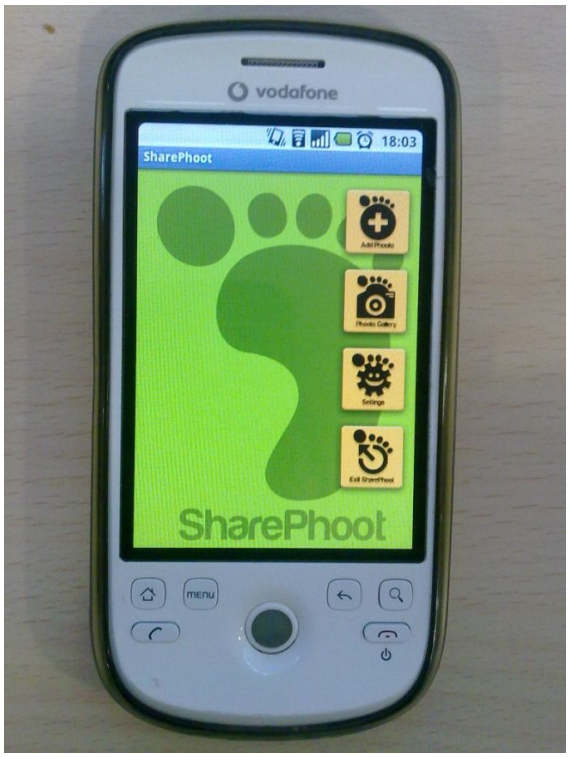


Figura 68: Menu de actividades de SharePhoot en el HTC

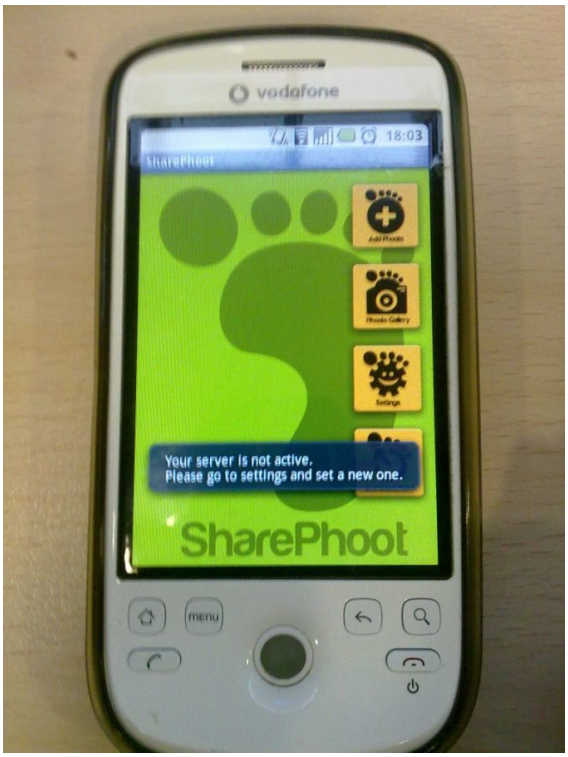


Figura 69: Tentativa errónea de conexión con ICE

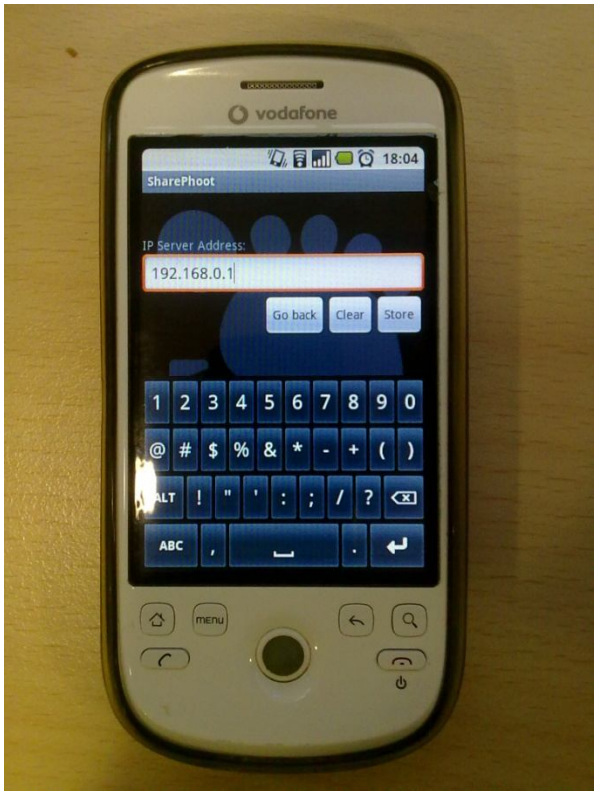


Figura 70: Pantalla de ajustes en el HTC

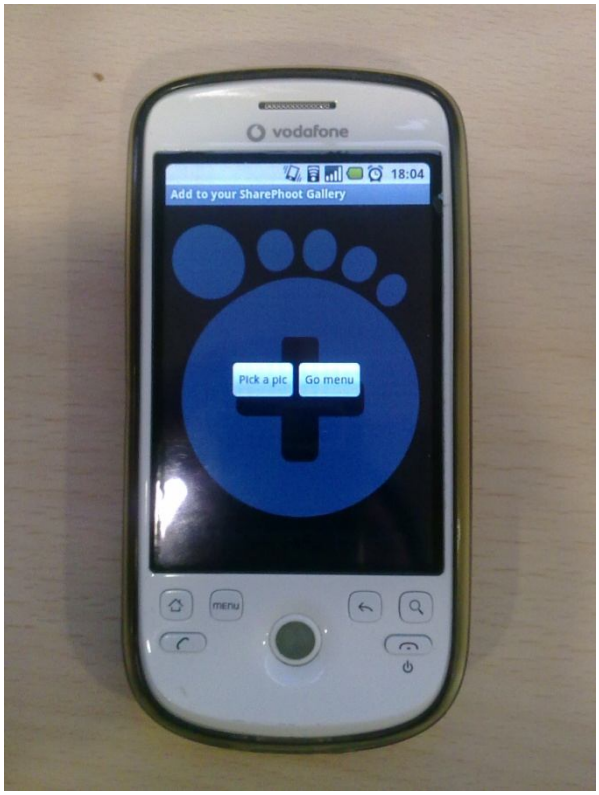


Figura 71:Pantalla del proceso de envío de fotos en el HTC

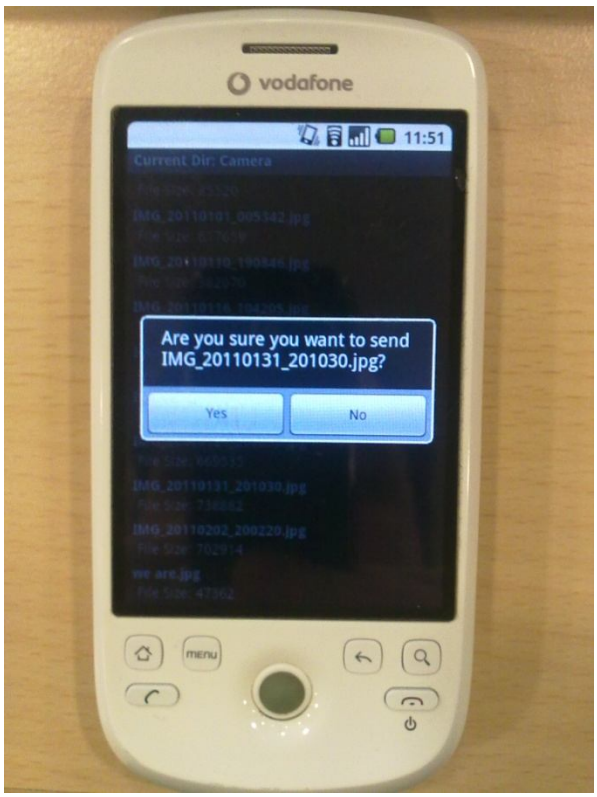


Figura 72: Pantalla de FileChooser, preguntando si el usuario está seguro de enviar dicha imagen

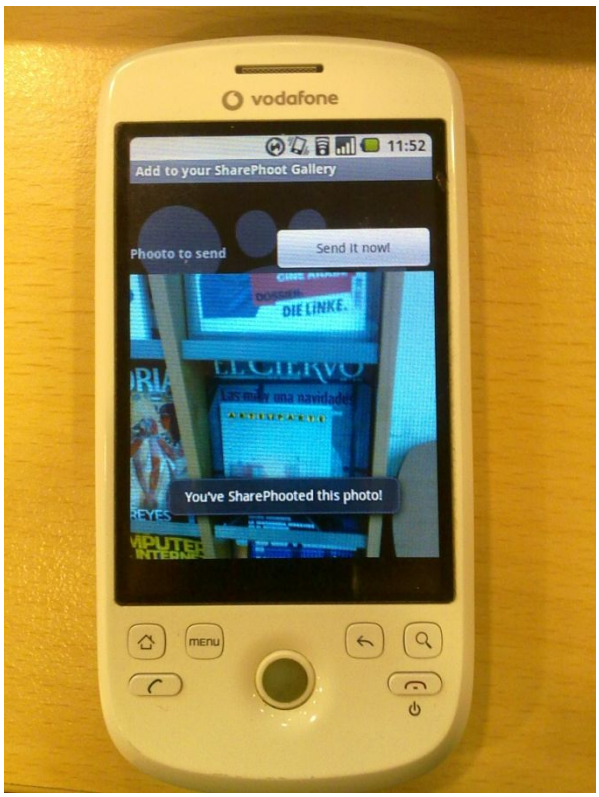


Figura 73: Momento exacto de envío de la imagen mostrada

Figura 74: Galería de fotos en el terminal HTC

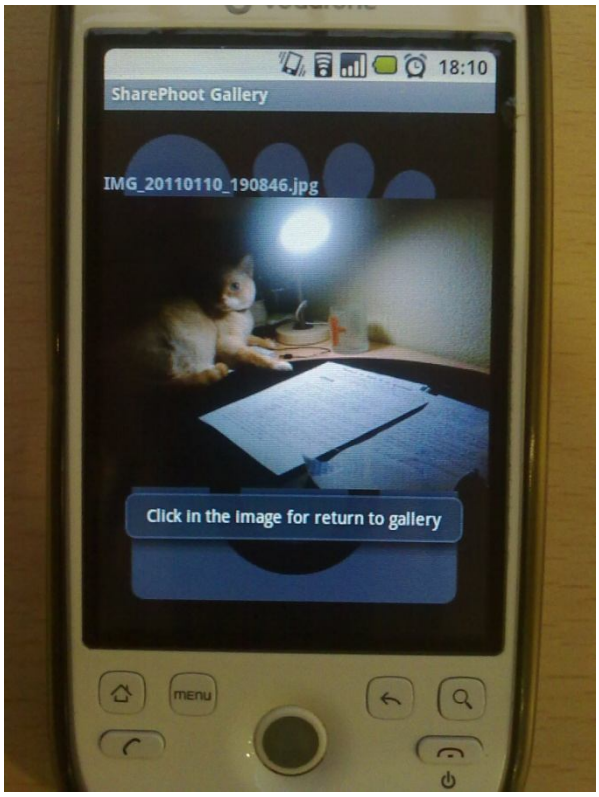


Figura 75: Pantalla de vista a tamaño original

Figura 76: Pantalla de confirmación al obtener del servidor la imagen de tamaño completo asociada a la miniatura

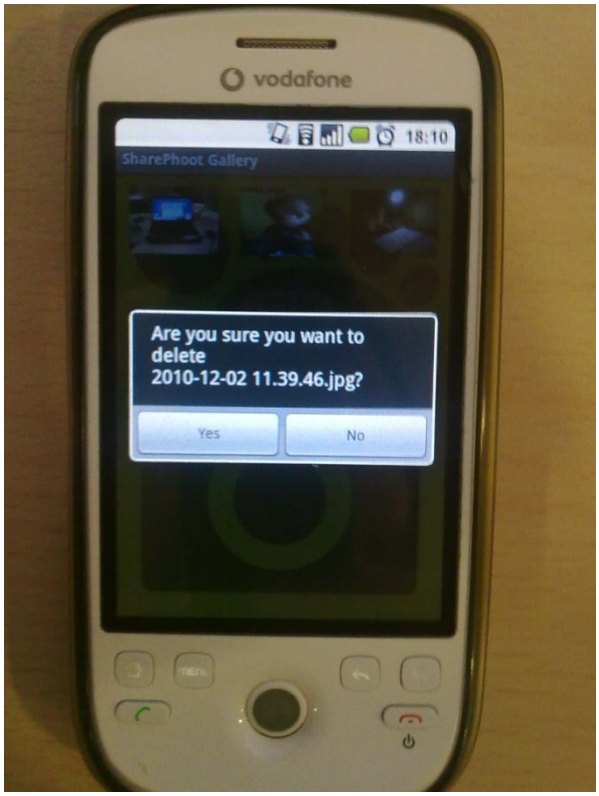


Figura 77: Pregunta de confirmación sobre la eliminación de una imagen en la HTC

5.3 Prueba de rendimiento

Se desarrolló una prueba de rendimiento a la aplicación **SharePhoot**, para probar la eficiencia del protocolo ICE cuando se mezcla con el sistema operativo de Google, Android.

5.3.1 Descripción de la prueba

Escenario

La prueba consiste en enviar **1, 10, 20, 30, 40 y 50** imágenes desde el servidor al cliente. Las imágenes enviadas son una representación en miniatura de la Figura 78. Se hicieron las pruebas con la misma imagen repetidas el número de veces que se requiriera, para una mayor simplicidad de los cálculos.

Figura 78: Imagen de la prueba de rendimiento

La parte del servidor fue realizada con el equipo descrito en el apartado **3.2**, un portátil *Acer Extensa 5635ZG* y las del cliente con el terminal descrito en el apartado **5.2.2**, un *HTC Magic*. Las propiedades de la línea *wireless* pública empleada como medio para conectarse, pueden verse en la Figura 79.

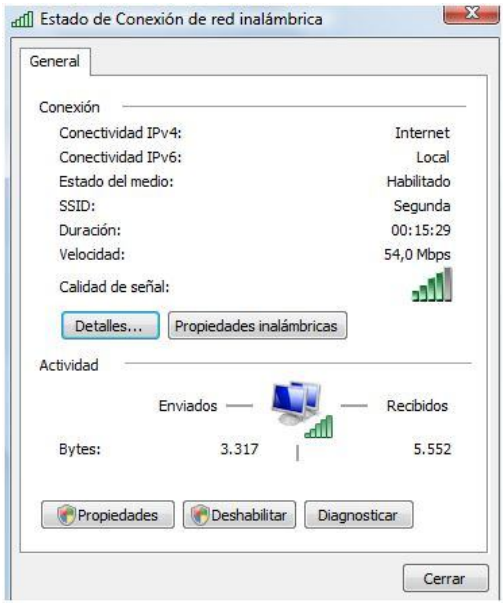


Figura 79: Conexión *wireless* pública empleada en la prueba

Funcionamiento de la prueba

Los pasos que sigue el protocolo para la petición de imágenes a través de **ICE** for Android, son los mostrados en la

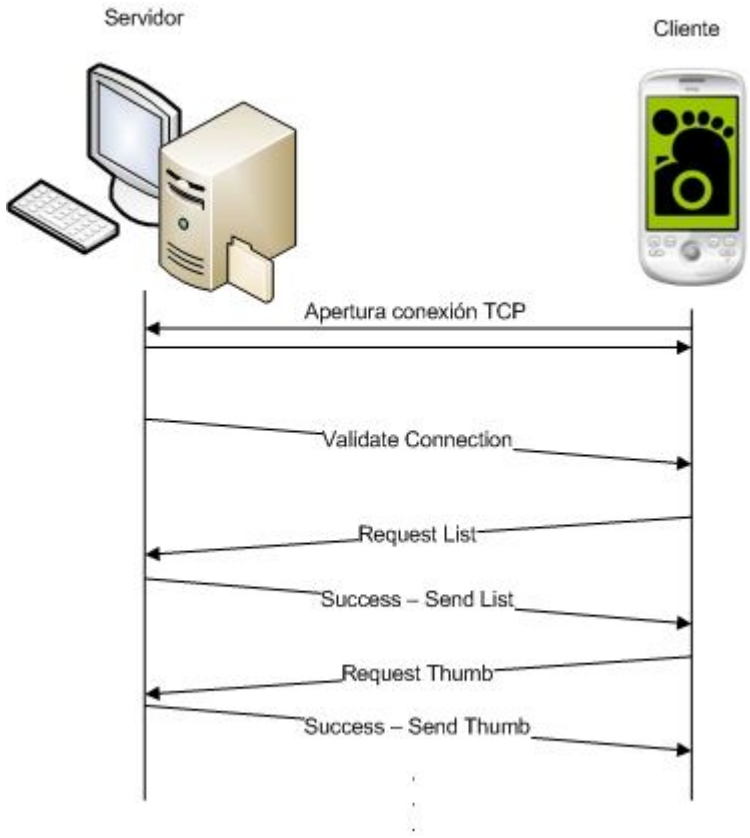


Figura 80: Procedimiento de petición de imágenes cliente-servidor

Los tiempos obtenidos han sido considerados desde el mensaje `Validate Connection` hasta el último `Success-Send Thumb`. El mensaje `Request-List`, es el necesario para descargar el fichero de texto donde se listan todos los archivos disponibles.

Tamaño de los datos:

- El tamaño de la miniatura mostrada en la Figura 78 es de **2826** *bytes*.
- El tamaño de la lista de archivos generada, aumenta a medida que aumentan el número de líneas que contiene. La evolución de los tamaños puede verse en la Tabla 19.
- El tamaño de un mensaje de tipo `request` para una única imagen, es de **71** *bytes*.
- El tamaño de un mensaje de tipo `success` contenedor de una miniatura es de **2856** *bytes*

Se puede concluir que el coste en bytes de pedir y recibir una miniatura, es de **2927** *bytes*.

Número de imagenes en la lista	Tamaño (bytes)
1	44
10	206
20	386
30	566
40	746
50	926

Tabla 19: Evolución del tamaño de la lista de archivos a descargar

5.3.2 Resultados obtenidos

Los resultados obtenidos en las 6 pruebas fueron coherentes en relación al tamaño de los datos enviados. Como se puede apreciar en el gráfico de la Figura 81, el resultado esperado es el mismo que el obtenido: aumento exponencial en función del aumento de imágenes.

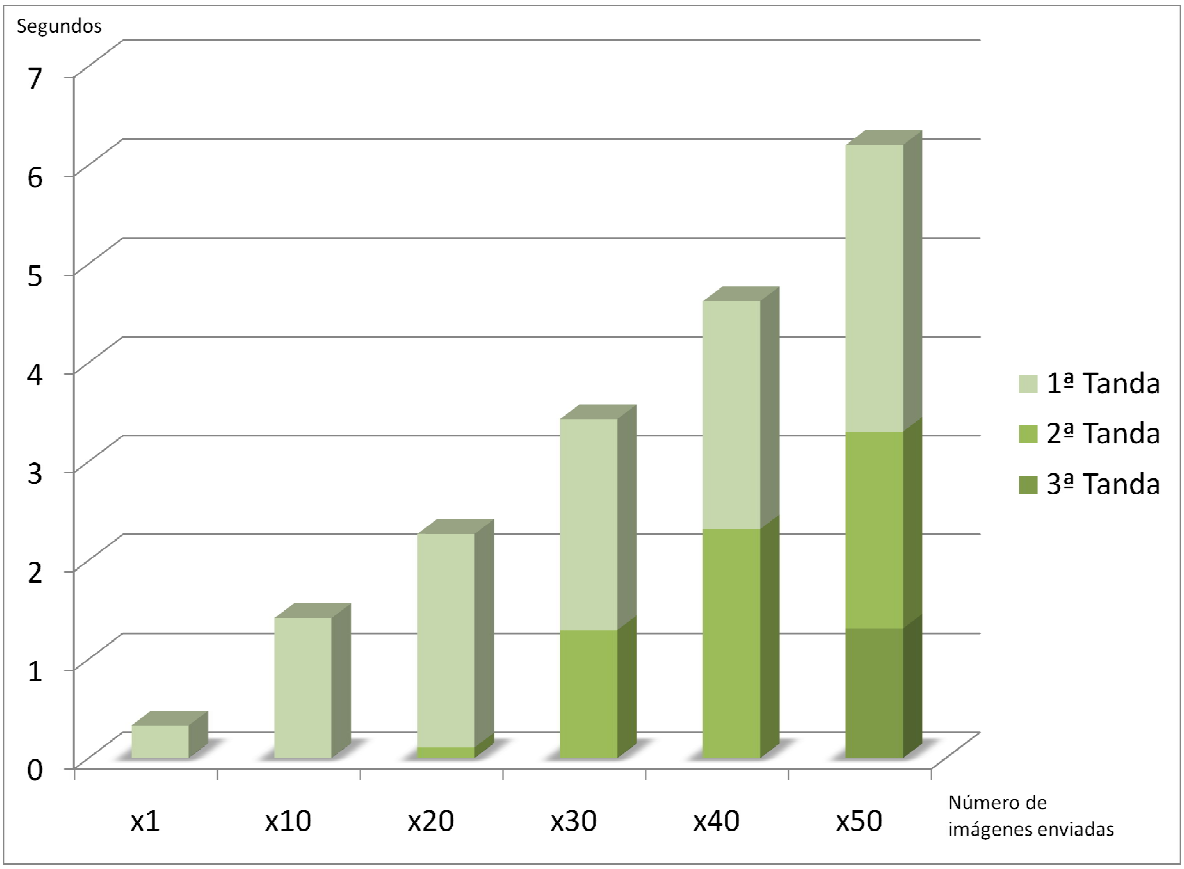


Figura 81: Tiempos obtenidos para cada prueba

Los datos se dividen en tandas, debido a que por razones de hardware, el tamaño de la pantalla del terminal empleado en la prueba, sólo acepta 18 miniaturas tal y como muestra la Figura 82.

Debido a la implementación de **SharePhoot**, ésta solo descarga las imágenes que ha mostrado una vez, consiguiendo así, un mayor rendimiento.

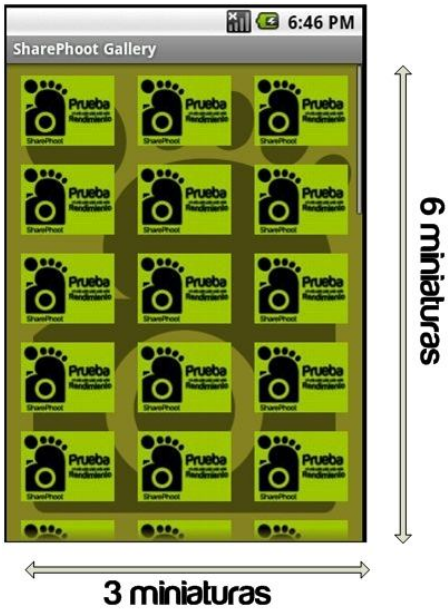


Figura 82: Pantalla del terminal

Se puede ver en las figuras Figura 83 y Figura 84 cómo es posible un envío de 1 o 10 miniaturas sin necesidad de más tandas dado que estarían dentro de los límites de la pantalla.

Figura 83: Mensajes de información del servidor para 1 miniatura

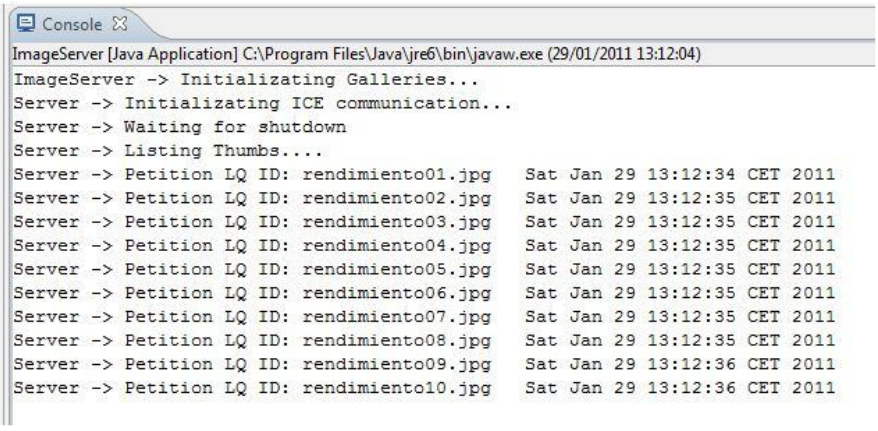


Figura 84: Mensajes de información del servidor para 10 miniaturas

Sin embargo, la Figura 85 muestra como al iniciar la petición de 20 imágenes, las miniaturas 19 y 20, se han enviado un segundo más tarde (que fue el tiempo que se tardó en desplazar la galería hacia abajo para ver más imágenes).

```
Server -> Listing Thumbs....
Server -> Petition LQ ID: rendimiento01.jpg   Sat Jan 29 13:22:28 CET 2011
Server -> Petition LQ ID: rendimiento02.jpg   Sat Jan 29 13:22:28 CET 2011
Server -> Petition LQ ID: rendimiento03.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento04.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento05.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento06.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento07.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento08.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento09.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento10.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento11.jpg   Sat Jan 29 13:22:29 CET 2011
Server -> Petition LQ ID: rendimiento12.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento13.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento14.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento15.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento16.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento17.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento18.jpg   Sat Jan 29 13:22:30 CET 2011
Server -> Petition LQ ID: rendimiento19.jpg   Sat Jan 29 13:22:31 CET 2011
Server -> Petition LQ ID: rendimiento20.jpg   Sat Jan 29 13:22:32 CET 2011
```

Figura 85: Mensajes de información del servidor para 20 miniaturas

Num. Imágenes	Descripción de los datos enviados	Tiempo en media(s)
x1	Primera tanda de 1 miniatura y lista de archivos	**0,329930**
x10	Primera tanda de 10 miniaturas y lista de archivos	**1,415353**
x20	Primera tanda de 18 miniaturas y lista de archivos	2,159635
	Segunda tanda de 2 miniaturas	0,106676
	Total	**2,266311**
x30	Primera tanda de 18 miniaturas y lista de archivos	2,135570
	Segunda tanda de 12 miniaturas	1,294128
	Total	**3,429698**
x40	Primera tanda de 18 miniaturas y lista de archivos	2,307051
	Segunda tanda de 22 miniaturas	2,315133
	Total	**4,622184**
x50	Primera tanda de 18 miniaturas y lista de archivos	2,900559
	Segunda tanda de 18 miniaturas	1,992355
	Tercera tanda de 14 miniaturas	1,310857
	Total	**6,203771**

Tabla 20: Desglose de tiempos obtenidos en las pruebas

Los datos desglosados por tandas se pueden consultar en la Tabla 20. Si se observa con detenimiento la tabla, se puede ver cómo ha sido posible un envío de **22** miniaturas. Esto es debido a que al hacer "*scroll*" en la aplicación, esta se quedó ligeramente bloqueada y aunque en el momento no mostrara por pantalla las imágenes, las peticiones pudieron hacerse secuencialmente.

Una vez finalizado el ciclo de vida de la *Activity* **Gallery**, se tendrá que volver a pedir todas las imágenes. Esto quiere decir, que cuando se inicia la Activity Gallery, ésta almacena las miniaturas que ha ido obteniendo, pero que al finalizar su ciclo de vida, éstas se pierden y se requiere volver a descargarlas.

5.4 Conclusiones

Se ha comprobado que la aplicación es compatible para ambas versiones de Android ya sea en el emulador que ofrece Google o en los terminales físicos.

Se ha comprobado también, que el emulador que ofrece Google es fiel a la experiencia obtenida en un terminal físico, pero que en una primera impresión, hace que no se tengan en cuenta cosas como el control de giro o el despliegue del teclado.

Tras la prueba de rendimiento, se puede concluir que el protocolo **ICE** para esta aplicación, es altamente eficiente dado que su eficiencia se cuantifica en un **96,55%** tal y como muestra la Ecuación 1. Cabe destacar que en el cálculo de los datos enviados, están incluidos los 71 bytes necesarios para formar el mensaje de `request`.

$$Eficiencia = \frac{Tamaño\ de\ la\ imagen}{Datos\ enviados} = \frac{2826\ bytes}{2927\ bytes} = 96{,}55\ \%$$

Ecuación 1: Eficiencia del protocolo ICE para una miniatura

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se indicarán las conclusiones obtenidas en la realización de todo el proyecto. Se indicarán las mejoras que se han ido anotando a medida que se ha ido analizando el código desarrollado, las futuras funcionalidades que puede implementar la aplicación y los objetivos logrados que fueron marcados al principio.

6.1 Mejoras de código

Dado que la versión explicada de **SharePhoot** en la memoria de este proyecto, es la **1.0**, se han encontrado varias mejoras de código que harían de la aplicación, una herramienta más eficiente en lo que se refiere a:

- Gestión de la memoria
- Mejora de la eficiencia del ancho de banda
- Mayor velocidad a la hora de generar galerías

Las mejoras de código propuestas son:

- Las peticiones de las miniaturas que genera **SharePhoot**, las genera una a una para cada vez que se muestra la galería. La mejora que se sugiere, es que pida de una vez todas las imágenes disponibles y las almacene temporalmente en la tarjeta **SD** del terminal, para aumentar en gran medida el rendimiento de la aplicación.

- Mientras que descarga imágenes, manda órdenes al servidor o se inicia la aplicación, se debiera mostrar una barra de progreso o algún mensaje informativo al usuario para evitar el malentendido de aplicación bloqueada.

- **SharePhoot**, tiene que enviar de una Activity a otra, el nombre bajo el cual se guardará la imagen en la tarjeta de memoria. Una posible mejora sería en la definición del fichero **SLICE**, un tipo de datos nuevo con dos atributos; nombre de la imagen y la imagen en cuestión.

- Cuando **SharePhoot** ordena borrar dos imágenes, hace dos peticiones; una para borrar la miniatura y otra para borrar la imagen a tamaño original. La mejora propuesta es que ambas peticiones se hicieran en una, modificando el código del servidor y la definición del fichero **SLICE**.

- Aislar en Activities separadas toda la funcionalidad de la aplicación. Esto quiere decir, por ejemplo, que la vista de ajustes pertenezca a una Activity independiente en vez de ser una vista adicional a principal de *SharePhootAct*.

- Correcciones que involucren una mejora de la eficiencia del almacenamiento de datos.

6.2 Trabajos futuros

En este momento se está trabajando en la versión **1.1** de **SharePhoot** para *bugs* y mejora de la eficiencia y las correcciones de código descritas en el anterior apartado 6.1, pero por el momento no se está implementando ninguna funcionalidad nueva.

Las nuevas funcionalidades, las cuales **SharePhoot** pudiera optar, entre otras, son:

- Dar la posibilidad de tomar imágenes de la cámara de fotos del terminal, dentro de la propia aplicación, y a su vez, una vez tomada dicha imagen, que esta se guardara directamente en la galería remota y en la tarjeta de memoria.

- Se sugiere que asociado a cada imagen, pudiera tener la opción de editar el nombre del archivo, añadir etiquetas y escribir comentarios acerca de esa imagen.

- Gracias al sistema **GPS** que tienen algunos modelos de Android, que a dichas etiquetas sugeridas en el punto anterior, se les pudiera añadir una marca de tiempo y lugar automáticamente.

- Una parte de ajustes más completa, donde se pudiera tener almacenada persistentemente una lista de servidores, se pudiera decidir si utilizar localización de fotos o no, o establecer manualmente un *timeout* para la conexión de servidores.

6.3 Objetivos logrados y conclusiones

Según los objetivos marcados desde un principio, se concluye lo siguiente:

Estudio del protocolo ICE y estudio de ICE en distintas plataformas

La realización de este proyecto, ha utilizado 3 sistemas operativos diferentes entre los cuales 2 versiones de Android, Linux y Windows.

Se ha estudiado a fondo el protocolo **ICE**, se han analizado distintos escenarios y se han probado en total 2 lenguajes de programación, **C++** y **Java**.

Se han desarrollado dos aplicaciones probando el funcionamiento de **ICE** en distintas plataformas, la primera aplicación descrita en el capítulo **3** y la segunda en el capítulo **4** con resultados satisfactorios.

Estudio de Android

El proceso de familiarización con Android, aunque en un principio fuera algo totalmente nuevo, se consiguió en poco tiempo. Esto reside en que el ciclo de vida de una aplicación Android es radicalmente diferente a cualquier programa en Java, pero con un poco de práctica, es altamente intuitivo.

Uno de los problemas que puede presentar Android como plataforma de desarrollo, es que requiere de un alto componente gráfico. Todo su desarrollo está basado en interfaces gráficas las cuales se pueden "adornar" de muchas maneras, y si no se sabe aprovechar bien esta característica, puede hacer que una aplicación muy eficiente, se convierta en poco atractiva al usuario final.

Estudio de ICE sobre Android

Se ha comprobado que es factible la utilización de **ICE** sobre terminales Android y que a nivel de rendimiento es eficiente.

Se han tenido no obstante numerosos problemas con la compatibilidad de versiones de **ICE**, perdiendo mucho tiempo en el arreglo de errores, dado que como ya se ha explicado, la versión actual de **ICE** for Android sólo es compatible con la versión **3.3.1**.

Implementación de una aplicación ICE sobre Android

Como se puede comprobar en el capítulo 4 y en el capítulo 5, se ha conseguido desarrollar una aplicación que implemente el sistema **ICE** sobre Android satisfactoriamente.

6.4 Otras conclusiones

Aparte de las mejoras propuestas para trabajos futuros o futuras versiones de **SharePhoot**, se puede recalcar, que la realización de este proyecto puede ayudar a estudios que involucren la tecnología middleware de comunicaciones **ICE** con Android.

El desarrollo de la aplicación de video surgió en un principio para portarla directamente a Android. La carencia de un ciclo de ejecución similar al de **SWING**, hizo que se rechazara esta idea y se transformara en lo que es ahora **SharePhoot**. Se decidió hacer una galería remota ya que el manejo de imágenes con Java y Android es bastante tedioso y difícil y se quiso aprovechar el que ya se estará familiarizado con ese tipo de archivos.

Otro de los "pros" de realizar una galería de imágenes remota, fue el que visualmente fuera muy atractivo para el usuario final el resultado de la ejecución de **SharePhoot**.

Bibliografía

1. **Varios**. Wikipedia. [En línea] 30 de 01 de 2011. http://es.wikipedia.org/wiki/Gadget.

2. **ZeroC.** Página principal de ZeroC. [En línea] [Consultado el: 30 de enero de 2011.] http://www.zeroc.com/.

3. —. ICE Overview. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/overview.html.

4. **Google.** Sitio Oficial de Android. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.android.com/.

5. **Apple.** Página del Apple Iphone. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.apple.com/es/iphone/.

6. **RIM.** Sitio oficial de Blackberry. [En línea] [Consultado el: 05 de 02 de 2011.] http://es.blackberry.com/.

7. **Google.** Licencia Apache. [En línea] [Consultado el: 16 de 01 de 2011.] http://source.android.com/source/licenses.html.

8. **Varios.** Sitio oficial de CORBA. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.corba.org.

9. **OHA.** Miembros de la OHA. [En línea] [Consultado el: 16 de 01 de 2011.] http://www.openhandsetalliance.com/oha_members.html.

10. **Solidsteel.** Movilpedia, definición de "Desvodafonizar". [En línea] [Consultado el: 05 de 02 de 2011.] http://www.movilpedia.es/nseries-de-gran-repercusion-nokia-n81/13868-desvodafonizar.html.

11. **Cosmin, Vasile.** Aplicaciones en Market y Appstore, Softpedia. [En línea] [Consultado el: 16 de 01 de 2011.] http://news.softpedia.com/es/Android-Market-tiene-130-000-de-aplicaciones-es-en-segundo-lugar-despues-de-Apple-App-Store-177516.html.

12. **Kondik, Steve.** Foros de Cyanogen Mods. [En línea] [Consultado el: 16 de 01 de 2011.] http://forum.cyanogenmod.com/files/category/1-stable-mod/.

13. **Google.** Distribución de Android a 4 de enero. [En línea] [Consultado el: 16 de 01 de 2011.] http://developer.android.com/resources/dashboard/platform-versions.html.

14. —. Android 2.2. [En línea] [Consultado el: 16 de 01 de 2011.] http://developer.android.com/sdk/android-2.2.html.

15. —. Android 2.3. [En línea] [Consultado el: 16 de 01 de 2011.] http://developer.android.com/sdk/android-2.3.html.

16. —. Screnszies in Android. [En línea] [Consultado el: 16 de 01 de 2011.] http://developer.android.com/resources/dashboard/screens.html.

17. —. What is Android. [En línea] [Consultado el: 05 de 02 de 2011.] http://developer.android.com/guide/basics/what-is-android.html.

18. —. Ciclo de actividades de Android. [En línea] [Consultado el: 18 de 01 de 2011.] http://developer.android.com/guide/topics/fundamentals.html#actlife.

19. **ZeroC.** Overview de ICE. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/ice.html.

20. **Spruiell, Mark.** Lanzamiento de ICE for Android. [En línea] [Consultado el: 16 de 01 de 2011.] http://www.zeroc.com/forums/announcements/4106-ice-android-0-1-0-slice2java-eclipse-plugin-released.html.

21. **ZeroC.** Download Page Ice For Android. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/labs/android/download.html.

22. **Spruiell, Mark.** Anuncio del lanzamiento de ICE for Android 0.1.1, Foros de ZeroC. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/forums/announcements/4279-ice-android-0-1-1-slice2java-eclipse-plug-3-3-1-released.html.

23. **Oracle.** JMF's Officials Site. [En línea] [Consultado el: 18 de 01 de 2011.] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html.

24. —. Attributions Site of JMF. [En línea] [Consultado el: 18 de 01 de 2011.] http://www.oracle.com/technetwork/java/javase/attributions-139785.html.

25. **ZeroC.** ICE Protocol. [En línea] [Consultado el: 21 de 01 de 2011.] http://www.zeroc.com/doc/Ice-3.4.1-IceTouch/manual/Protocol.html.

26. —. ICE Manual. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/doc/Ice-3.4.1-IceTouch/manual/.

27. —. Manual de ICE, Ejemplo de Marshaling. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.zeroc.com/doc/Ice-3.4.1-IceTouch/manual/Protocol.38.2.html.

28. **VMWARE.** Vmware Official Site. [En línea] [Consultado el: 05 de 02 de 2011.] http://www.vmware.com/es/.

29. **Ubuntu.** Página de descargas de Ubuntu 9.10. [En línea] [Consultado el: 05 de 02 de 2011.] http://releases.ubuntu.com/karmic/.

30. **Wireshark.** Página oficial de Wireshark. [En línea] [Consultado el: 24 de 01 de 2011.] http://www.wireshark.org/download.html.

31. **Hispanic, Washington.** Privacidad en Facebook. [En línea] [Consultado el: 21 de 01 de 2011.] http://www.washingtonhispanic.com/nota6280.html1.

32. **Google.** Hello Views. [En línea] [Consultado el: 21 de 01 de 2011.] http://developer.android.com/guide/tutorials/views/index.html.

33. **H3R3T1C.** Foros de dream.in.code. [En línea] [Consultado el: 21 de 01 de 2011.] http://www.dreamincode.net/forums/topic/190013-creating-simple-file-chooser/.

34. **ZeroC.** RPM's válidas para ICE. [En línea] [Consultado el: 28 de 01 de 2011.] http://www.zeroc.com/download.html#linux_rpms.

35. —. Paquete de REDHAT para ICE. [En línea] [Consultado el: 28 de 01 de 2011.] http://www.zeroc.com/download/Ice/3.4/Ice-3.4.1-rhel5-x86_64-rpm.tar.gz.

Presupuesto

Actividades del proyecto y duración

Las tareas realizadas durante el proyecto, se pueden dividir en 3 actividades: parte de **ICE**, parte de Android y la redacción de la memoria. La duración **total** de las tareas está situada en las 3 primeras barras del gráfico. Las barras de 30 días o menos, son las tareas desglosadas.

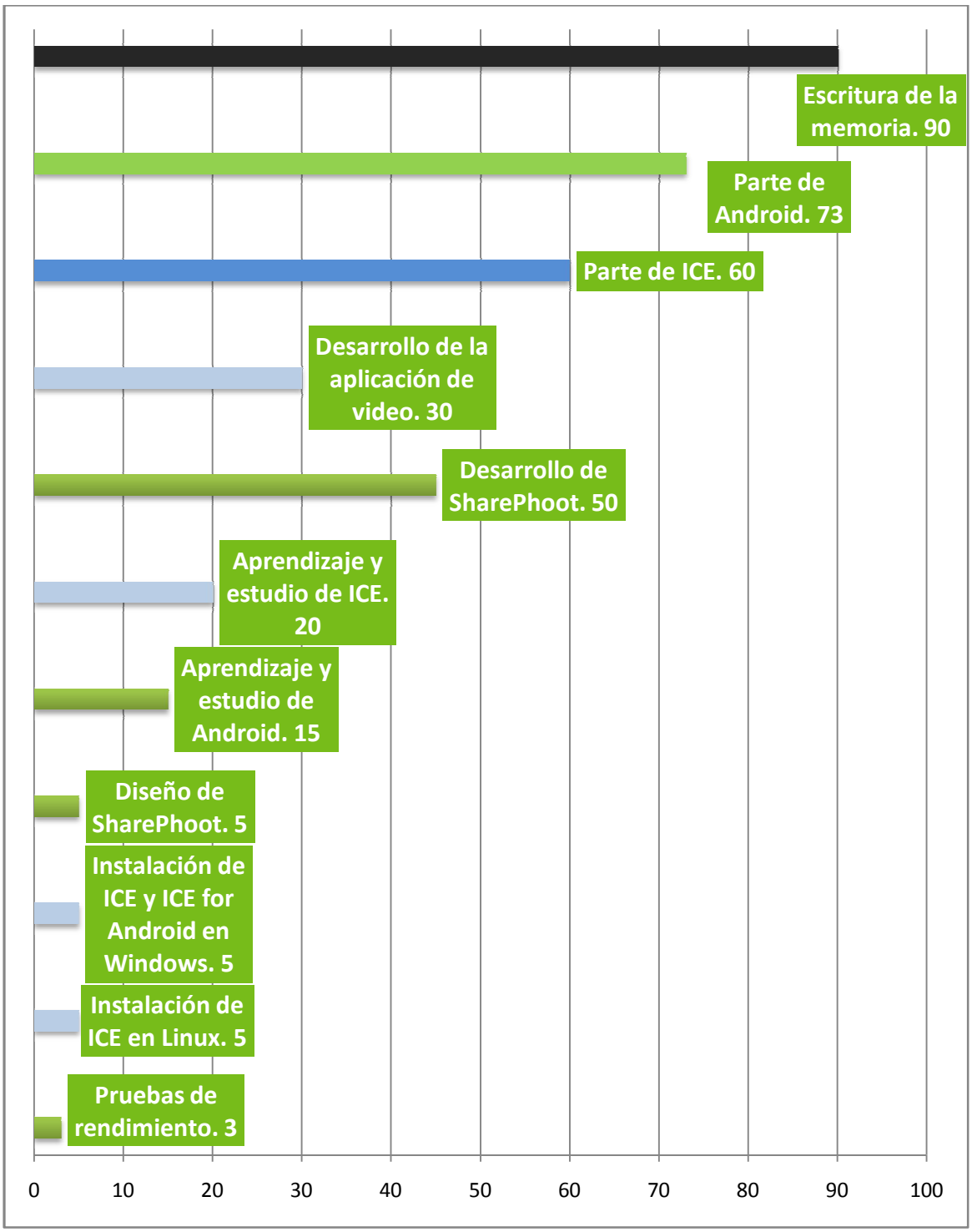


Figura 86: Grafico de duración de tareas

Presupuesto de proyecto

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

1.- Autor:

Israel Cendrero Sánchez

2.- Departamento:

Ingeniería Telemática

3.- Descripción del Proyecto:

- Titulo
- Duración (meses) **9**

Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

27.094,00 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	Categoría	Dedicación (hombres mes) [a)]	Coste hombre mes	Coste (Euro)
Estévez Ayres, Iria Manuela	Ingeniero Senior	0,5	4.289,54	2.144,77
Cendrero Sánchez, Israel	Ingeniero	7,5	2.694,39	20.207,93
	Hombres mes	**8**	**Total**	**22.352,70**

[a)] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d)]
Terminal HTC	200,00	100	5	60	16,67
Equipo portátil	700,00	100	9	60	105,00
Disco de Backup	150,00	100	9	60	22,50
Google's Dev Phone($399)	289,82	100	3	60	14,49
				Total	**158,66**

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Licencia Photoshop CS1	Adobe	49,00
Licencia Developer Android ($25)	Google	18,11
	Total	67,11

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,…

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	22.353
Amortización	159
Subcontratación de tareas	0
Costes de funcionamiento	67
Costes Indirectos	4.516
Total	27.094

Nota:

Hay en el presupuesto dos valores que han sido convertidos de la divisa USD($) a EUR(€) se hizo a día 02/02/2011 con un valor monetario de:

1,00 € = 1,38$

Los valores cambiados son la licencia de desarrollador Google ($25) y el terminal *Google's Developers Phone 1* ($399).

Anexo 1

Instalación de ICE 3.3.1 y el plug-in Slice2Java para Eclipse en Windows

Anexo 1: Instalación de ICE 3.3.1 y el plug-in Slice2Java para Eclipse en Windows

Para instalar la última distribución de la tecnología **ICE** en Windows, no conlleva más que seguir una serie de sencillos pasos que puede consultar en la misma página web de **ZeroC**. Aún así, este anexo resume los pasos a seguir y las consideraciones que hay que tener para poder usar **ICE** en una máquina corriendo en Windows.

Para poder usar la tecnología **ICE**, necesita, aparte de los drivers, librerías y utilidades de las que necesitamos para el lenguaje en el cual va a desarrollar (recuerde que puede utilizarse en **C++, Java, .NET, Python, PHP, Ruby y Objetive-C**) necesita tener una distribución de **C++** instalada en su máquina, ya que esta tecnología se encuentra compilada en ella.

Este aparente problema se soluciona instalando cualquiera de las distribuciones gratuitas de drivers para Visual Studio que dispone de la página oficial de Microsoft. Para este anexo se han utilizado en concreto los de la distribución de 2005 en su versión **SP1** la cual puede encontrarse en la siguiente dirección.

http://msdn.microsoft.com/vstudio/support/vs2005sp1/default.aspx

Una vez tenga la distribución de **C++** instalada, puede proceder a descargar la última distribución de Ice publicada. Se descarga en formato *Microsoft Installer* y no hay más que seguir los pasos.

http://www.zeroc.com/download.html

Una vez tenga instalado todo, puede pasar a la instalación de una de las herramientas más útiles en el desarrollo de aplicaciones **Ice en Java: Eclipse Slice2Java Plug-in**.

Eclipse es uno de los *Integrated Development Environment* (**IDE**) más extendido y usado por los programadores de Java. Así pues, tiene que tenerlo descargado y corriendo en su equipo. Una cosa hay que tener en cuenta, y es que necesitará (si quiere soporte oficial) la versión **3.4 Ganymede** de éste.

Descargue Eclipse de su página oficial siguiendo el enlace a continuación.

http://www.eclipse.org/downloads/download.php

Y una vez todo listo, tiene que hacer lo siguiente para instalar **Slice2Java** en su distribución de Eclipse:

1. Se hace click en `Help` → `Software Updates`
2. Después, en la pestaña `Available Software`, hacemos click en `Add site…`
3. En Location, añada la dirección `http://www.zeroc.com/download/eclipse`
4. Seleccione `Slice2Java Plug-In` e instale.

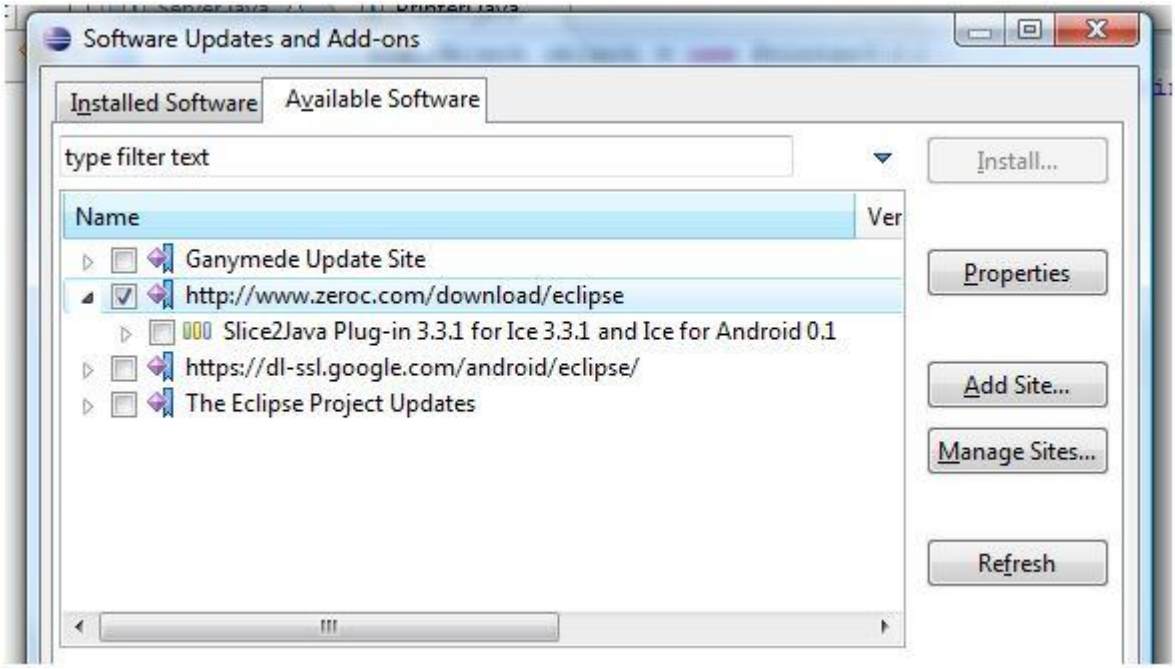


Figura 87: Ventana de Software Updates de Eclipse.

Una vez hechos estos pasos ya tendrá listo su constructor de ficheros automático de las diferentes clases necesarias para hacer funcionar una aplicación sobre **ICE**.

Anexo 2

Instalación de ICE 3.4.1 en Ubuntu 9.10

Anexo 2: Instalación de ICE 3.4.1 en Ubuntu 9.10

Uno de los grandes inconvenientes que tiene la tecnología **ICE**, es que aunque esta soporta Linux, no está diseñada para una de las distribuciones más extendidas a nivel de usuario, **Debian**[REF Debian]. En este anexo se explica cómo instalar **ICE** en esta distribución, más concretamente **Ubuntu 9.10**. Para obtener la distribución de **ICE** la cual es posible instalar en Debian, es la **Red Hat Enterprise Linux 5.4 x86_64 RPMs (34)**.

Para instalar **ICE** en Ubuntu, se requieren las dependencias del paquete **Berkeley Database 4.6.21** el cual tiene que estar instalado con anterioridad**.** También se necesitan adicionalmente, las dependencias de **BZIP2** y **EXPAT**. Todos estos paquetes pueden ser instalados a través del siguiente comando:

```
$apt get install <nombre_del_paquete>
```

Para poder convertir los paquetes de extensión **.rpm** (Versión de **RedHat**) en paquetes auto instalables de debían (**.deb**) se requiere del programa **alien** el cual puede ser instalada de la misma manera que las dependencias:

```
$apt get install alien
```

Una vez todo listo, se comienza a convertir los paquetes **.rpm** descompridos del fichero **Ice-3.4.1-rhel5-x86_64-rpm.tar.gz (35)** en paquetes **.deb** con el siguiente comando:

```
$alien –k <nombre_del_paquete>
```

Una vez se hayan convertido todos los paquetes mediante el comando `alien`, se debería tener en **.deb** los paquetes que se muestran en la Figura 88.

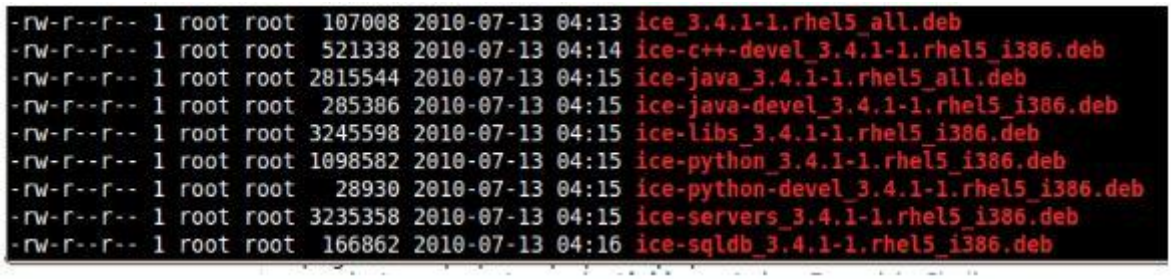


Figura 88: Paquetes RedHat convertidos para Debian

Una vez convertidos todos los paquetes, se instalarán uno a uno con el siguiente comando, de la manera que se puede apreciar en la Figura 89:

```
$sudo dpkg –I <nombre_del_paquete>
```

El orden en el cual han de ser instalados es crítico, ya que unos paquetes dependen de otros, y si no se hace en este orden, la instalación no podrá ser realizada. El orden a seguir es el siguiente:

- **ice-3.4.1-1.rhel5_all.rpm**
- **ice-C++-devel-3.4.1-1. rhel5.i386.rpm**
- **ice-libs-3.4.1-1. rhel5. i386.rpm**
- **ice-servers-3.4.1-1. rhel5. i386.rpm**
- **ice-utils-3.4.1-1. rhel5. i386.rpm**
- **ice-java-3.4.1-1.rhel5_all.rpm**
- **ice-java-devel-3.4.1-1.rhel5. i386.rpm**
- **ice-python-3.4.1-1.rhel5_all.rpm**
- **ice-python-devel-3.4.1-1.rhel5. i386.rpm**
- **ice-sqldb-3.4.1-1.rhel5_all.rpm**

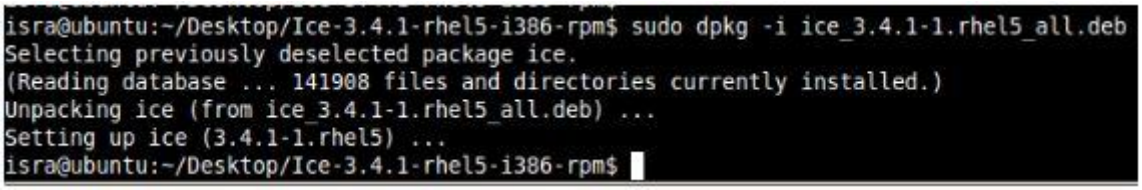


Figura 89: Instalación de paquetes en Debian

Cuando ya se hayan instalado todos los paquetes en el orden referenciado, estará lista para usar la distribución de **ICE** en el sistema **Debian**.

Anexo 3

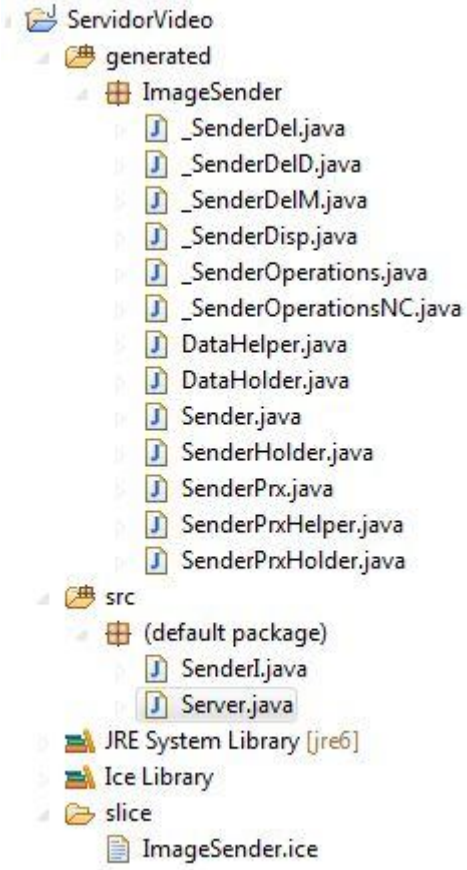
Descripción del plug-in Slice2Java para Eclipse

Anexo 3: Descripción del *plug-in Slice2Java* para Eclipse

En este anexo, describiremos el significado y el funcionamiento de las carpetas que se crean al añadir un constructor de **Slice2Java** a nuestro proyecto de **ICE**.

Para poder convertir el proyecto en un proyecto **ICE**, solo se tendrá que hacer *click* derecho en el nombre del proyecto, y añadir un constructor de **Slice2Java**. Con esto, el proyecto en Java se convertirá en un proyecto de **ICE** sobre Java dejándolo preparado para ser utilizado como tal.

Figura 90: Cómo añadir un constructor Slice en Eclipse



Una vez añadido el constructor, aparecerán en el proyecto las carpetas "*generated*" y "*slice*". Así como la referencia a la librería externa "*Ice Library*".

Para empezar a desarrollar código **ICE**, no es necesario que se escriba cada una de las clases que se pueden ver dentro del paquete *ImageSender*. Esa pila de clases se generan automáticamente en el momento que definamos un módulo Ice dentro de la carpeta *slice*.

Cuando se defina un módulo en **ICE**, se traduce al instante en un nuevo paquete. Cuando se defina una nueva interfaz, esta genera todas las clases necesarias para su uso. En este caso se puede ver como la interfaz "*Sender*" genera todos los archivos que comienzan por "_", las clases objeto que son *Sender* y *SenderHolder* y todas aquellas relacionadas con el proxy de comunicación. Las clases *DataHelper* y *DataHolder*, se generan al definir un nuevo tipo de variable de tipo *array* de *bytes*.

En resumen, en el momento que un fichero de la extensión "**.ice**" aparezca en la carpeta *slice*, generará todo lo necesario para no tener que escribir ni una sola línea adicional de la aplicación, siempre y cuando el fichero *slice* esté bien formado.

Anexo 4

Manual de SharePhoot

Anexo 4: Manual de SharePhoot

Aunque el uso de **SharePhoot** es completamente intuitivo e informa al usuario en todo momento de lo que tiene que hacer o cómo lo puede hacer, se presenta aquí el manual de usuario y las operaciones que se pueden realizar.

SharePhoot
Manual de usuario

Añadir Fotos (*Add Phooto*)

Figura 91: Cómo acceder a Añadir fotos

Para acceder al proceso para añadir fotos a la galería compartida, pulse en "*Add Phooto*"

Figura 92: Cómo comenzar con el proceso de subida

Si quiere continuar con el proceso, pulse en "*Pick a Pic*" o si quiere volver al principal, pulse "*Go menú*"

Figura 93: Cómo seleccionar una foto

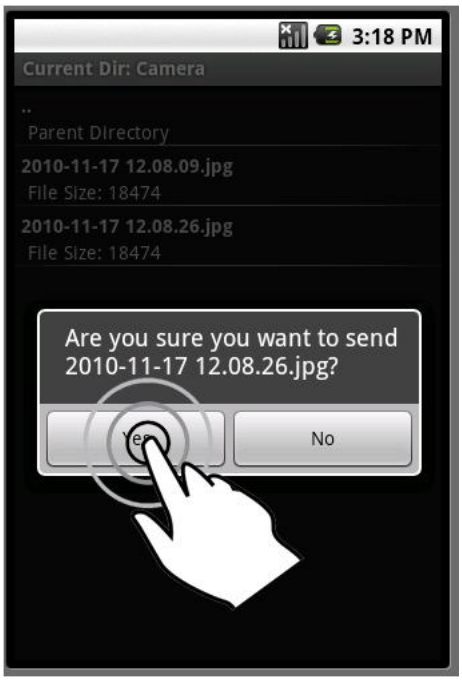


Figura 94: Mensaje de confirmación para subir una foto

Una vez seleccionada la imagen que quiere añadir a la galería remota, pulse encima del nombre de dicha imagen

Confirme la operación en caso de ser correcta.

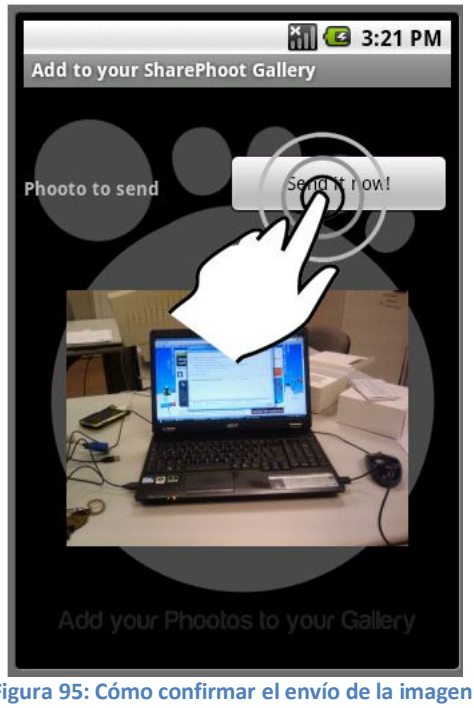


Figura 95: Cómo confirmar el envío de la imagen

Una vez obtenga la pantalla de confirmación, pulse en "*Send it Now!*" y si se realiza la operación satisfactoriamente, obtendrá un mensaje positivo.

Galería de fotos (*Phooto Gallery*)

Figura 96: Cómo acceder a la galería

Para acceder a la galería compartida, ha de pulsar en el icono al igual que muestra la Figura 91

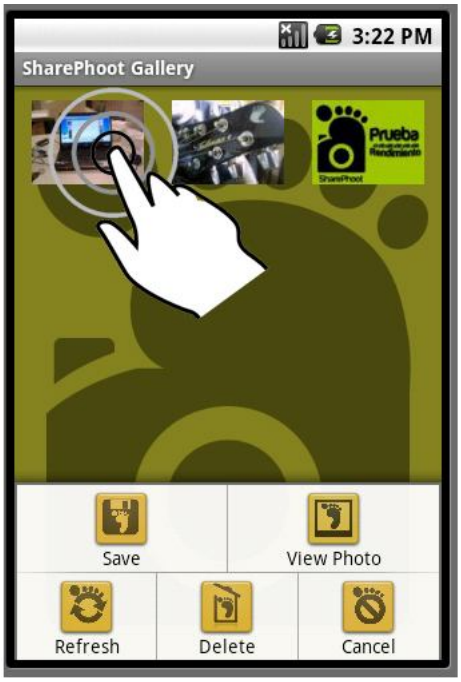


Figura 97: Cómo ineractuar con las imagenes

Seleccione la foto con la que quiere operar. Podrá al aparecer el menú: Guardarla en su teléfono (*Save*), Verla más grande(View) Refrescar la galería (*Refresh*), borrarla (*Delete*) o cancelar la operación

Figura 98: Cómo volver a la galería

Si decide ver la imagen más grande, para volver a la galería, deberá pulsar en la imagen.

Figura 99: Cómo borrar una imagen

Si desea eliminar una imagen, deberá de pulsar sobre ella, y confirmar la operación de eliminación.

Ajustes (*Settings*)

Figura 100: Cómo acceder a los ajustes

Para acceder a los ajustes, ha de pulsar en el icono al igual que muestra la Figura 91

Figura 101: Cómo introducir una IP

Para añadir la dirección **IP** del servidor, ha de pulsar en el campo de texto para que aparezca el teclado.

Figura 102: Cómo guardar una IP de un servidor ICE

Con el teclado desplegado, escriba una dirección **IP** bien formada y de un servidor válido y pulse *Store*.

Figura 103: Mensaje de confirmación de IP

Si ha escrito una **IP** bien formada y válida, se guardará en el sistema y podrá acceder a su galería **SharePhoot**.

Salir (*Exit*)

Figura 104: Cómo salir de SharePhoot

Para salir de SharePhoot, pulse *Exit*.

Anexo 5

Código servidor de imágenes

Anexo 5: Código servidor de imágenes

Clase GalleryHandler:

Imports:

```
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;

//Almacena las direcciones completas de las imágenes de calidad alta
private  static String[] listFullPathHQ = null;

//Almacena los nombres de las imágenes independientemente de su calidad
private  static String[] listFiles = null;


//Inicia los atributos de la clase GalleryHandler
public static void init(){

//Creamos nuevo directorio de la carpeta contenedora de imágenes de resolución completa.
      File directory = new File("../ImageServer/hq_images");

     //Obtenemos los nombres de las imágenes del directorio e inicializamos el array contenedor
     de las direcciones completas de todas las imágenes.
      listFiles = directory.list();
      listFullPathHQ = new String[listFiles.length];

       //Vamos rellenando uno a uno las direcciones completas de los archivos para luego poder
       generar lo mismo pero con las miniaturas.
      for (int i = 0; i < listFiles.length; i++) {
            listFullPathHQ[i] = "../ImageServer/hq_images" + "/" +
                                listFiles[i];
      }
}


//Método que se encarga de generar la galería de miniaturas
public static boolean resizeGallery() {

[...]

//A la hora de crear un nuevo BufferedImage, se debe indicar el tamaño de ésta, cualidad que se
aprovecha para poner un tamaño pequeño acorde con una galería en un dispositivo móvil.
BufferedImage img = new BufferedImage(100,75,BufferedImage.TYPE_INT_RGB);

//Con drawImage(), se le asigna como argumento un nuevo fichero (el cual es la dirección
completa de la imagen a tamaño real) y aprovechamos la encapsulación que nos proporciona Java
para obtener una instancia nueva del mismo tamaño el cual queremos para nuestra galería de
miniaturas. A modo de resumen, con createGraphics y drawImage, rellenamos los bytes necesarios
```

para obtener una miniatura que luego guardar en memoria.

//Con el parámetro Image.SCALE_SMOOTH nos aseguramos que la miniatura tenga una compresión suave y no de dientes de sierra. Esto crea una estética mejor de nuestra galería de miniaturas.

```
img.createGraphics().drawImage(
                        ImageIO.read(new File(listFullPathHQ[i]))
                        .getScaledInstance(100, 75, Image.SCALE_SMOOTH),0,
                                                0, null);
```

//Método estático write de ImageIO el cual, indicándole un Buffer de imagen (img), un tipo de archivo, y un fichero con la dirección donde quiere guardarse (en este caso, el directorio de miniaturas), genera una imagen nueva con los ajustes que anteriormente se hayan hecho.

```
ImageIO.write(img, "JPG", new File("../ImageServer/lq_images",
                                    listFiles[i]));

[...]


public static void refresh(){

      //Borramos todas las miniaturas antiguas una a una.
      File directory = new File("../ImageServer/lq_images");
      for (int i = 0; i < directory.list().length; i++) {
            File toDelete = new File( "../ImageServer/lq_images/"
                                        +directory.list()[i]);
            toDelete.delete();
      }
      //Volvemos a generar la galería de miniaturas.
      init();
      resizeGallery();
}
```

Clase **ImageServerI**:

Imports:

```
import java.awt.image.BufferedImage;
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Date;
```

//La clase padre de Sharephoot._ImageServerDisp es abstracta y contiene los métodos definidos en el archivo Sharephoot.ice.

```
public class ImageServerI extends Sharephoot._ImageServerDisp
```

Con esta definición del interfaz, tendremos que reescribir los métodos que se definieron

anteriormente dado que su herencia lo requiere. Los métodos a reescribir son los siguientes:

- `StringList getThumbList();`
- `Image getThumb(string id);`
- `Image getHQImg(string id);`
- `void sendName(string name);`
- `void deleteImage(string name);`
- `void sendImage(Image img);`

Método **StringList getThumbList()**:

```
//Método que devuelve el array de nombres de todos las imágenes del servidor
public String[] getThumbList(Current __current) {

            System.out.println("Server -> Listing Thumbs.... ");
            //Simplemente se devuelve el listado de ficheros que se encuentra de manera
            estática en un atributo en la clase myGallery
            return ImageServer.myGallery.getListFiles();
      }
```

Método **Image getThumb(string id)**:

```
//Método que devuelve una miniatura correspondiente al nombre que se le indica como parámetro
public byte[] getThumb(String id, Current __current) {
            System.out.println("Server -> Petition LQ "+"ID: "+id+"
"+new Date());
            String path = "../ImageServer/lq_images/";
            byte[] image = null;
            try {

                  //Abrimos el fichero que nos indican por parámetro del directorio de
                  miniaturas
                  File file = new File(path+id);
                  //Lo convertimos a un array de bytes y lo devolvemos
                  image = imageToByteArray(file);
            } catch (IOException e) {
                  e.printStackTrace();
            }

            return image;
      }
```

Método **Image getHQImage(string id)**:

```
//Método que devuelve la foto a tamaño real según el nombre que se le indica como parámetro
public byte[] getThumb(String id, Current __current) {
            System.out.println("Server -> Petition LQ "+"ID: "+id+"
"+new Date());
            String path = "../ImageServer/lq_images/";
            byte[] image = null;
            try {
```

```
                //Abrimos el fichero que nos indican por parámetro del directorio de
                imágenes en calidad alta.
                File file = new File(path+id);
                //Lo convertimos a un array de bytes y lo devolvemos
                image = imageToByteArray(file);
            } catch (IOException e) {
                e.printStackTrace();
            }

            return image;
}
```

Método **void sendName(string name);**

```
public void sendName(String name, Current __current) {
            //Guarda el nombre del fichero a guardar
            actualName = name;
      }
```

Método **deleteImage(string name);**

Con este método se podrán borrar remotamente imágenes simplemente con indicarles el nombre.

```
public void deleteImage(String name, Current __current) {

      System.out.println("Server-> Deleting this photo: "+name);

      //Se abren los ficheros que se van a eliminar
      File toDeleteLQ = new File("../ImageServer/lq_images/"+name);
      File toDeleteHQ = new File("../ImageServer/hq_images/"+name);
      //Se eliminan
      toDeleteLQ.delete();
      toDeleteHQ.delete();
      //Se actualiza la galería
      GalleryHandler.refresh();
}
```

Método **void sendImage(Image img);**

Por último, este método se encargará de recoger los bytes que formarán la imagen final y refrescará la galería para que esté actualizada.

```
      public void sendImage(byte[] img,Current __current) {

[...]
            //Inicialización de variables y creación de un flujo de bytes para poder formar la
            imagen.
            InputStream in = new ByteArrayInputStream(img);
            BufferedImage image
```

```
            image = ImageIO.read(in);
            //Guardamos en el directorio de imágenes de calidad alta
            ImageIO.write(image, "JPG",
                                  new File("../ImageServer/hq_images/"
                                              +actualName));

[...]
            //Una vez tengamos la imagen recibida y guardada en el directorio de calidad alta,
            procedemos a actualizar la galería
            actualName=null;
            GalleryHandler.refresh();
      }
```

Método **byte[] imageToByteArray(File file);**

```
public byte[] imageToByteArray(File file) throws IOException {

      byte[] image = null;
      //Se abren los buffers necesarios para poder crear un array de bytes
      FileInputStream in = new FileInputStream(file);
      BufferedInputStream reader = new BufferedInputStream(in);
      //Se crea el array de bytes del mismo tamaño que los bytes disponibles(en este caso la
      imagen es única y no hay más datos en espera)
      int length = reader.available();
      image = new byte[length];
      //Se rellena el array con los bytes de la imagen y se cierra el bufer.
      reader.read(image, 0, length);
      reader.close();
      return image;

}
```

Clase **ImageServer**:

```
public class ImageServer {

      //Necesidad de este tipo de atributo estático ya que es requerido por ImageServerI para su
      manejo de imágenes
      static GalleryHandler myGallery;

      public static void main(String[] args) {

      //Inicialización de variables
      [...]
            Ice.Communicator ic = null;

            //Se crea un nuevo GalleryHandler, se inicia y se pasa a generar la galería
            myGallery = new GalleryHandler();
            myGallery.init();
            myGallery.resizeGallery();
```

```
        //El objeto Communicator de ICE, se inicializa con los argumentos que se le pueden
        pasar como parámetros. En este caso no se han requerido ninguno.
        ic = Ice.Util.initialize(args);

        //Se crea un adaptador ICE el cual se encarga de establecer los parámetros
        necesarios para la comunicación. "ImageServerBind" será el identificador de red por
        el que se buscará en el lado del cliente. Al no especificar ninguna dirección ip, se
        establecerá una escucha en todos los adaptadores de red que existan, así las
        peticiones pueden llegar por cualquier puerta de enlace. Se establecerá una
        conexión TCP en el puerto 60000.
        Ice.ObjectAdapter adapter = ic.createObjectAdapterWithEndpoints
                                    ("ImageServerBind", "tcp -p 60000");

        //Se genera un Nuevo objeto de tipo ImageServerI que será el cual tendrá la
        información de los métodos que se pueden llamar remotamente.
        Ice.Object iceObject = new ImageServerI();

        //Al adaptador que se creó anteriormente, añadimos la identidad de nuestro
        servidor al objeto de tipo ImageServerI y lo dejamos listo para activar.
        adapter.add(iceObject, ic.stringToIdentity("ImageServerBind"));

        //En este punto de la ejecución del servidor es donde se lanza el servidor y pasa a un
        segundo plano con todos los ajustes que previamente se han establecido.
        adapter.activate();

        //El objeto Communicator quedará a la escucha por si hay algún problema con la
        conexión o por si se le pide que deje de funcionar.
        ic.waitForShutdown();

    //El resto del código consiste en el manejo de posibles excepciones generadas por el intento
    de acceder a la red o por malfuncionamiento en general
    [...]
}
```

Anexo 6

Código de SharePhoot

Anexo 6: Código de SharePhoot

Activity **SharePhootAct:**

Imports:

```
import java.util.regex.Pattern;
import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.inputmethod.InputMethodManager;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
```

Atributos:

```
//Campo de texto donde se recogerá la ip del servidor
EditText ipEditText;

//Ip del servidor
static String ipServer = "";

//Boolean que evitará el lanzar la vista de bienvenida si se reinicia la aplicación.
private boolean init = true;

//Objeto proxy de ICE el cual nos permitirá hacer las peticiones al servidor en esta Activity para verificar la validez de la conexión.
Sharephoot.ImageServerPrx communicator;
```

Método **onCreate(Bundle savedInstanceState):**

Este es el método más importante de toda la aplicación, puesto que es el que se encarga de arrancar la aplicación entera (aunque no todos sus módulos). Es sencillo ya que se ha decidido separar el código en distintos métodos para hacer más fácil su comprensión.

```
//Al llamar automáticamente a este método (lo llama el propio terminal) le indicamos el constructor de la clase padre para hacerlo visible.
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    //Si es la primera vez que ha sido llamado, lanzamos primero la pantalla de bienvenida, si no, simplemente lanzamos la del menú de operaciones
    if(init){
            //Mostramos la pantalla de bienvenida e indicamos que ya se ha mostrado
            setContentView(R.layout.main);
            init=false;
```

```
            //Con esta llamada asíncrona, a los 3 segundos lanzamos el menú de operaciones e
            inicializamos todos los parámetros necesarios para que la aplicación empiece a
            funcionar.

            //La razón de lanzarlo 3 segundos más tarde, es para que dé tiempo a ver la
            pantalla de bienvenida. Si no se hiciera de esta manera, saldría el menú de
            operaciones directamente.
            new Handler().postDelayed(new Runnable() { public void run() {
                  initLauncher();
            } }, 3000);
      }else{
        initLauncher();
      }
}

//Método que se encarga de la inicialización de la pantalla del menú de operaciones
private void initLauncher(){

      //Se muestra la pantalla del menú de operaciones
      setContentView(R.layout.menu_launcher);

      //Inicializamos todos los botones buscando las imágenes en el índice de identidades
      Button buttonGallery = (Button)findViewById(R.id.launcher_gallery);
      Button buttonAdd = (Button)findViewById(R.id.launcher_add);
      Button buttonSettings=(Button)findViewById(R.id.launcher_settings);
      Button buttonExit = (Button)findViewById(R.id.launcher_exit);

      //Una vez creados los botones, se les añade un escuchador de eventos para cuando sean
      presionados.
      buttonGallery.setOnClickListener(buttonGalleryListener);
      buttonAdd.setOnClickListener(buttonAddListener);
      buttonSettings.setOnClickListener(buttonSettingsListener);
      buttonExit.setOnClickListener(buttonExitListener);
}
```

Metodos OnClickListener para los botones del menú de operaciones

```
//Atributo y redefinición del método onClick para el botón que lanza la Activity Gallery
private OnClickListener buttonGalleryListener = new OnClickListener()
{
    public void onClick(View v)
    {

      //En este objeto Bundle, es donde se pasarán argumentos entre una Activity y otra. La
      activity llamada los recogerá indicándole la etiqueta del parámetro que busca.
      Bundle bundle = new Bundle();
//En la etiqueta ipIceServer se encontrará la ip del servidor ICE al cual hacer peticiones.
      bundle.putString("ipIceServer", ipServer);

//Se crea un nuevo Intent
```

```
    Intent intent = new Intent();
//Se indica la clase "llamadora" y la clase objetivo
    intent.setClass(SharePhootAct.this, Gallery.class);
//En el context se le añaden los parámetros anteriormente creados
    intent.putExtras(bundle);
//Y por fin se lanza la nueva actividad quedando esta en segundo plano
    startActivity(intent);
    }
};

//Atributo y redefinición del método onClick para el botón que abre la Activity AddToGallery para añadir una foto a la galería
private OnClickListener buttonAddListener = new OnClickListener()
{

    //En esta redefinición del método onClick se procederá de la misma manera que en el onClick del escuchador buttonGalleryListener ya que el procedimiento es exactamente igual a este solo que con la diferencia de que la Activity objetivo es AddToGallery
    public void onClick(View v)
    {
     Bundle bundle = new Bundle();
     bundle.putString("ipIceServer", ipServer);

     Intent intent = new Intent();
     intent.setClass(SharePhootAct.this, AddToGallery.class);
     intent.putExtras(bundle);
     startActivity(intent);

    }
};

//Atributo y redefinición del método onClick para el botón que lanza la vista de los ajustes del servidor ICE.
private OnClickListener buttonSettingsListener = new OnClickListener()
{
    public void onClick(View v)
    {
            //En este onClick() solo se requiere lanzar la vista de los ajustes, ya no fue necesario desarrollar una activity nueva para este fin. Su funcionamiento se explicará a continuación
            initSettings();
    }
};

//Atributo y redefinición del método onClick para el botón que cierra la aplicación.
private OnClickListener buttonExitListener = new OnClickListener()
{
    public void onClick(View v)
    {
      //Mostramos un mensaje de despedida en la pantalla y finalizamos el ciclo de vida de esta Activity principal.
      Toast.makeText(SharePhootAct.this,"Bye :) !",
                     Toast.LENGTH_SHORT).show();
```

```
        finish();
    }
};
```

Vista de ajustes

El método initSettings() se encarga de lanzar la vista de los ajustes y de inicializar todos los botones y campos de dicha vista.

```
private void initSettings(){

        setContentView(R.layout.settings);

        //Creamos los botones para volver a la vista principal, para limpiar el campo de texto y para
        comprobar y guardar la ip
        Button buttonGoBak = (Button)findViewById(R.id.back);
        Button buttonClearSettings = (Button)findViewById(R.id.clear);
        Button buttonStore = (Button)findViewById(R.id.store);

        //Añadimos los escuchadores de eventos(cuando se presionan dichos botones)
        buttonGoBak.setOnClickListener(buttonGoBackListener);
        buttonClearSettings.setOnClickListener(buttonClearSettingsListener)
        buttonStore.setOnClickListener(buttonStoreListener);

        //Campo de texto donde irá la ip a comprobar
        ipEditText = (EditText) findViewById(R.id.ip_server);
        ipEditText.setText("");
}

//Atributo y redefinición del método onClick para el botón que vuelve a la vista de operaciones.
private OnClickListener buttonGoBackListener = new OnClickListener()
{
    public void onClick(View v)
    {
            //Relanzamos el inicializador de la vista de operaciones (es decir, volvemos a pintar
            la vista y los botones del menú de operaciones
            initLauncher();
    }
};

//Atributo y redefinición del método onClick para el botón limpia el campo dónde se recoge la ip
del servidor IP.
private OnClickListener buttonClearSettingsListener = new
OnClickListener()
{
    public void onClick(View v)
    {
      //Simplemente se pone el campo con un valor vacío.
      ipEditText.setText("");
    }
};
```

```
private OnClickListener buttonStoreListener = new OnClickListener()
{
    public void onClick(View v)
    {

[...]
      //El objeto InpudMethodManager imm se crea para que una vez se termine de editar el
      texto, el teclado que aparece al presionar el campo de texto, vuelva a su estado original y
      no quede estático tapando toda la vista, ya que sólo queremos que aparezca a la hora de
      introducir una ip.
      InputMethodManager imm=(InputMethodManager)
                              getSystemService(Context.INPUT_METHOD_SERVICE)
      imm.hideSoftInputFromWindow(ipEditText.getWindowToken(), 0);

      //En el momento que presione "Store" , esconderemos el teclado (como hemos hecho en la
      acción anterior) y obtendremos la cadena de texto que contenga en ese momento.
      String tempIP = (ipEditText.getText()).toString();

     //Tenemos que validar si es una cadena de texto del formato:
      [0-255]. [0-255]. [0-255]. [0-255] de lo que se encarga el método validateIP y que nos
     devuelve un booleano indicando el resultado
      if(validateIP(tempIP)){

            //Pero no es suficiente ya que puede ser una dirección ip no válida para la conexión
            ICE. Es por eso por lo que se llama al  método testConnection el cual lanzará una
            excepción de tipo Ice.LocalException y se manejará de forma que muestre un
            mensaje indicando de la no validez de esa ip e instando al usuario a escribir una
            válida.
            testConnection(tempIP);
            //Si ha pasado este punto, es evidente que la ip es válida y tiene conectividad así
            que se procede a almacenar la ip que va a ser fija en todo el desarrollo de la
            aplicación
            ipServer = tempIP;

            //Muestra un mensaje de confirmación y tras 2 segundos y medio para dar tiempo
            al usuario que lea el mensaje, vuelve a pintar la vista del menú de operaciones
            dejándolo listo para añadir fotos o acceder a la galería
            Toast.makeText(SharePhootAct.this, "Ip Stored",
                                              Toast.LENGTH_SHORT).show();
             new Handler().postDelayed(new Runnable() { public void run()
             {
                  initLauncher();
             } }, 2500);
//Se muestran mensajes de error si no se cumplen las condiciones (Ip malformada o ip no válida) y
termina el método y la declaración del atributo del escuchador
[...]
```

Metodos auxiliares de detección de fallos

```
//Tras una cadena de texto que en teoría es una ip pasada como argumento, se utilizan expresiones regulares para comprobar el formato de esa cadena de texto
private boolean validateIP(String iPaddress){
      final Pattern IP_PATTERN =
            Pattern.compile("^([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                        "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                        "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
                        "([01]?\\d\\d?|2[0-4]\\d|25[0-5])$");
      return IP_PATTERN.matcher(iPaddress).find();
}

//Método que lanza una excepción (realmente él no, el API de ICE) si la conexión no ha podido realizarse con un servidor/objeto ICE. De esta manera se puede controlar la conexión de tal manera que no es necesario estar esperando largos tiempos en los cuales la aplicación queda completamente congelada.
public void testConnection(String ip) throws Ice.LocalException{
      //Se crean los objetos de conexión necesarios para establecer la conexión ICE como se ha explicado en el capítulo 3  distinto
      Ice.Communicator ic = Ice.Util.initialize();
      //En este caso, se modifica el tiempo de timeout (-t <ms>) ya que se van a realizar tareas de rol de cliente, así pues se necesita comprobar que dicha conexión es factible. Con un tiempo de 500ms en un entorno local valdría para realizar un ping y comprobar el estado. Como mejora o trabajos futuros, esta implementación se podría mejorar incorporando algún formulario o añadiendo algún campo en la vista de ajustes para indicar el tiempo que el usuario está dispuesto a esperar para ver si la conexión es válida. Dado al carácter didáctico de la aplicación se ha ignorado este asunto.
      Ice.ObjectPrx base = ic.stringToProxy("ImageServerBind:tcp -h "
                                            + ip+ " -t 500 -p 60000");
      Sharephoot.ImageServerPrx tester = Sharephoot.ImageServerPrxHelper.
                                          checkedCast(base);
      //Se realize el ping y si todo va bien no lanza ninguna excepción confirmando así la disponibilidad de la conexión
      tester.ice_ping();
}
```

Activity **Gallery:**

Imports:

```
import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.GridView;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
import android.widget.AdapterView.OnItemClickListener;
```

Atributos:

```
//Este atributo sirve para identificar si una imagen ha sido seleccionada o no, en el caso de que sea distinto a -1, será la id de la imagen
static int selection = -1;
//ImageAdapter con el cual generaremos la vista de la galería
static ImageAdapter ia;
//Ip del servidor ICE
static String ip;
//Variable que nos sirve para evitar que se utilice el menú en el caso de estar fuera de la galería
static boolean lock = false;
//Objeto Proxy de ICE que se usará para eliminar imágenes de la galería
Sharephoot.ImageServerPrx deleter;
```

Método **onCreate(Bundle savedInstanceState):**

```
public void onCreate(Bundle savedInstanceState) {

[...]
      //Tras llamar al constructor de la clase padre, se obtiene el XML correspondiente al de la galería. Ya que es importante tener creado los elementos en el momento en el que se vaya a establecer las miniaturas
      setContentView(R.layout.gallery_main);
      GridView gridview = (GridView) findViewById(R.id.gridview);

      //Objeto Bundle el cual se encarga de obtener los parámetros que previamente ha enviado la Activity anterior . En este caso, la IP para el servidor ICE
      Bundle bundle = this.getIntent().getExtras();
```

```
        ip = bundle.getString("ipIceServer");

        //Este es un punto crítico, puesto que a la hora de crear un nuevo ImageAdapter, éste,
        intentará establecer la conexión ICE. Si lo consigue todo irá como lo esperado, pero si no,
        lanzará una excepción de tipo Ice.LocalException y la Activity de Gallery no llegará nunca a
        mostrarse, puesto que al atrapar la excepción se mostrará un mensaje de error y se dará
        por finalizada esta Activity, no pasando nunca de la pantalla de operaciones.
        ia = new ImageAdapter(this,ip);
        //Si ha creado exitosamente el objeto ImageAdapter, está listo para mostrarse en pantalla
        como una galería perfectamente formada.
        gridview.setAdapter(ia);

//Manejo de las excepciones
[...]
        //Se añade a la vista un escuchador de eventos al igual que se hizo con los botones de la
        Activity anterior. Esto se hace para detectar que un usuario ha presionado o seleccionado
        una imagen.
        gridview.setOnItemClickListener(new OnItemClickListener() {
         //Al igual que con los botones, en este se tiene que reescribir el método que se encarga de
         ejecutar acciones cuando es llamado.
         public void onItemClick(AdapterView<?> parent, View v, int
         position,long id) {
              //Cuando se ha pulsado o seleccionado una imagen, se debe obtener la id de dicha
              imagen y aparte, mostrar el menú de opciones dejándolo listo para que el usuario
              elija qué hacer con ella.
              selection = position;
              openOptionsMenu();
//Cierre de llaves
[...]
```

Como consecuencia de crear un menú de opciones, se tienen que reescribir dos métodos fundamentales, uno para crearlo, y otro para decidir qué hacer con la opción seleccionada.

```
//Cuando se pulsa la tecla menú, o se llama al método openOptionsMenu(), Android internamente
llama a este método el cual se encarga de "inflar" o dicho de otra manera, mostrar por la parte de
debajo de la pantalla el menú de opciones.
public boolean onCreateOptionsMenu(Menu menu) {
        //Optiene los elementos necesarios para "inflarlo". Entre esos elementos, se encuentra el
        fichero XML sharephoot_menu.xml, el cual se explicará en el anexo 7, el cual se encarga de
        mostrar los iconos del menú.
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.sharephoot_menu, menu);
        return true;
}
```

El otro método (**onOptionsItemSelected**(MenuItem item) no es más que un decisor switch en el cual tras pasarle la identidad del elemento pulsado o elegido se decide qué hacer. Todo el formato del switch será el siguiente a excepción de la opción de borrar, pero eso se explicará más adelante.

```
//Se obtiene el objeto el cual ha sido pulsado
switch (item.getItemId()) {
    //Id del element seleccionado
    case R.id.ic_save:
      //Si está dentro de la galería
      if(!lock){
            //Acción elegida
      }else{
            //Mensaje de error indicando que el menú ha sido abierto en otra vista o Activity
            fuera de la galería
      }
      //Una vez terminada la acción, se deja la selección por defecto (ninguna)
      selection=-1;
      return true;
```

Guardar (Save)**:**

```
private void savePhoto(){
      //Si se ha seleccionado alguna foto
      if(selection!=-1){
            //Se crea un nuevo objeto Bundle para enviarle a la Activity que se va a encargar de
            guardar la foto, el array de bytes que compone la misma y su nombre.
            Bundle bundle = new Bundle();
            //Se pide mediante el método getHQImg(id) los bytes de la imagen que se quiere
            guardar, al objeto ImageAdapter común a toda esta Activity.
            byte[] requestedPhoto = ia.myServer.getHQImg(
                                    ia.list[selection]);
            //Se añaden los datos al objeto Bundle
            bundle.putByteArray("photo", requestedPhoto);
            bundle.putString("name", ia.list[selection]);

            //Y como es común ya, se lanza una nueva Activity sin cerrar la actual. En este caso
            es la Activity SaveThisPhoto, la cual se explicará más adelante.
            Intent intent = new Intent();
            intent.setClass(Gallery.this, SaveThisPhoto.class);
            intent.putExtras(bundle);
            startActivity(intent);
//Si no se cumplen las condiciones mostramos mensaje de error
[...]
```

Ver (View Photo):

```
private void viewPhoto(){
      //Si se ha seleccionado alguna foto
      if(selection!=-1){
            //Se muestra la vista donde va a ir la imagen en grande y se crea el elemento donde
            va a ir esa imagen (un ImageView)
            setContentView(R.layout.view_photo);
            ImageView imgView = (ImageView) findViewById(R.id.photo_full);

            //Aparte, se crea también una etiqueta para el nombre de la imagen.
```

```
            TextView tv = (TextView)findViewById(R.id.label_title);
            tv.setText(ia.list[selection]);

            //Se crea un objeto Bitmap el cual será el que contenga los bytes de la foto que
            previamente hemos obtenido mediante el objeto común ImageAdapter de la misma
            manera que se hizo en savePhoto()
            Bitmap loadedImage = null;
            byte[] photo = ia.myServer.getHQImg(ia.list[selection]);
            loadedImage = BitmapFactory.decodeByteArray(photo, 0,
                                                        photo.length);
            //Una vez todos los elementos creados, se muestran en pantalla y aparte, se
            muestra el mensaje que indica al usuario que para voler debe presionar la imagen
            imgView.setImageBitmap(loadedImage);
            Toast.makeText(this, "Click in the image for return to
                                  gallery", Toast.LENGTH_SHORT).show();

            //Y para finalizar, se añade un escuchador de eventos al igual que en los botones
            para detectar cuando el usuario presione la imagen. Cuando haga esto, solo tendrá
            que llamar al método refresh(), que reconstruirá la galería otra vez.
            imgView.setOnClickListener(new OnClickListener(){
                  public void onClick(View arg0) {
                        refresh();
                  }
            });
//Si no se cumplen las condiciones mostramos mensaje de error
[...]
```

Refrescar (Refresh)**:**

```
private void refresh(){
      //Como se hizo en SharePhootAct,tendremos que enviar a la nueva Activity(en este caso, a
      la misma, pero una instancia nueva) la IP del sevidor ICE del cual tomar las imágenes.
      Bundle bundleRefresh = new Bundle();
      bundleRefresh.putString("ipIceServer", ip);

      //Esto se hace para evitar confusión con la selección de la imagen.
      if(selection!=-1)selection=-1;

      //Se anadem los parámetros, se establece la clase lanzadora y la que va a ser lanzada
      (como se ha explicado, en este caso son la misma) y se finaliza la instancia actual de la
      Activity.
      Intent intentRefresh = new Intent();
      intentRefresh.setClass(Gallery.this, Gallery.class);
      intentRefresh.putExtras(bundleRefresh);
      startActivity(intentRefresh);
      finish();
}
```

Eliminar (Delete):

```
//Se crea un constructor de diálogos de alerta
AlertDialog.Builder builder = new AlertDialog.Builder(this);

//A este constructor de diálogos de alerta se le añade un mensaje, la no posibilidad de cancelar,
una acción para la respuesta positiva y otra acción para la respuesta negativa.
builder.setMessage("Are you sure you want to
        delete\n"+ia.list[selection]+"?")
       .setCancelable(false)
       .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {

           //Si la respuesta es afirmativa (el usuario quiere borrar la imagen del servidor) se
           hace la petición al servidor ICE para que borre dicha imagen.
           ia.myServer.deleteImage(ia.list[selection]);
           //Dejamos la selección por defecto y refrescamos la galería.
           selection=-1;
           refresh();
            }
       })
       .setNegativeButton("No", new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {

                //En cambio si la respuesta es negativa (el usuario no quiere eliminar la
                imagen del servidor) se descarta este diálogo.
                dialog.cancel();
           }
       });
//Se crea el diálogo de alerta por fin y se muestra dejando en segundo plano todo hasta que se
conteste a la pregunta.
AlertDialog alert = builder.create();
alert.show();
```

Cancelar (Cancel):

En cancelar sólo se llamará al método finish() de Activity para cerrar el menú y la galería y volver a la vista de operaciones principal.

Clase **ImageAdapter:**

Imports:

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.GridView;
import android.widget.ImageView;
```

Esta clase es la que se encarga de generar la vista de la galería. Es también, la que se encarga de establecer la conexión con el servidor **ICE** y hacerle las peticiones.

Al ser un objeto de uso común, es muy útil que se le definan de manera estática los métodos que se comunican con el servidor, ya que así no es necesario sobrecargar la red con aperturas de conexión múltiples.

Atributos importantes:

```
//Objeto que será compartido por toda la Activity de la galería para hacer peticiones al servidor.
Sharephoot.ImageServerPrx myServer;
//Lista de imágenes que se encuentran en el servidor remoto.
static String[] list;

//En el constructor se le pasará el contexto en el que se encuentra y la ip del servidor.
public ImageAdapter(Context c,String ip) throws Ice.LocalException {
      [...]
      //Se inicia la conexión con el servidor, si falla, lanzará la excepción de tipo
      Ice.LocalException que será recogida por la Activity de la galería y ésta decidirá qué hacer
      con ella. Tras la conexión, se accede a conocer el total de las imágenes que se tienen en el
      servidor.
      initIceConnection(false);
      list = myServer.getThumbList();
}


public View getView(int position, View convertView, ViewGroup parent) {
[...]

//A la vista, se le añaden los ajustes que se crean oportunos, tales como márgenes, tipo de
escalado de la imagen, o tamaño de cada cuadrícula
        imageView = new ImageView(mContext);
        imageView.setLayoutParams(new GridView.LayoutParams(100, 75));
        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        imageView.setPadding(8, 8, 8, 8);
    }
[...]
```

```
        //Se obtiene la imagen del servidor mediante la petición getThumb (para obtener la
        miniatura de ella, se descodifica el array de bytes que la forma y se le asigna al Bitmap el
        cual irá insertado en la vista.
        byte[] image = null;
        image = myServer.getThumb(list[position]);
        Bitmap loadedImage = null;
        loadedImage = BitmapFactory.decodeByteArray(image, 0,
                                                    image.length);
        imageView.setImageBitmap(loadedImage);

//Será llamado automáticamente hasta que estén todas las imágenes en la cuadrícula
    return imageView;
}
```

Activity **SaveThisPhoto:**

Imports:

```
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import android.app.Activity;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Bitmap.CompressFormat;
import android.os.Bundle;
import android.os.Handler;
import android.widget.ImageView;
import android.widget.Toast;
```

Atributos:

```
static ImageView imgView;
```

Método **onCreate(Bundle savedInstanceState):**

```
public void onCreate(Bundle savedInstanceState) {

      super.onCreate(savedInstanceState);
      setContentView(R.layout.save_this_photo);

      Bundle bundle = this.getIntent().getExtras();

      byte[] photo = bundle.getByteArray("photo");
      String name = bundle.getString("name");

      imgView = (ImageView) findViewById(R.id.vistaImagen);
      Bitmap loadedImage = null;
      loadedImage = BitmapFactory.decodeByteArray(photo, 0,
```

```
      photo.length);
      imgView.setImageBitmap(loadedImage);
      imgView.invalidate();

      if (StoreByteImage(this, photo, 100, name)){
            Toast.makeText(this, "Image saved.",
      Toast.LENGTH_SHORT).show();
          new Handler().postDelayed(new Runnable() { public void run() {
                finish();
           } }, 2000);
      }
```

Método **StoreByteImage(params):**

```
public static boolean StoreByteImage(Context mContext, byte[] imageData,
int quality, String expName) {

File sdImageMainDirectory = new File("/sdcard/sharephoot");
FileOutputStream fileOutputStream = null;
String nameFile = expName;
try {

      Bitmap myImage = BitmapFactory.decodeByteArray(imageData, 0,
                imageData.length);


      fileOutputStream = new FileOutputStream(
                sdImageMainDirectory.toString() +"/" + nameFile );


                BufferedOutputStream bos = new BufferedOutputStream(
                          fileOutputStream);

                myImage.compress(CompressFormat.JPEG, quality, bos);

                bos.flush();
                bos.close();

           } catch (FileNotFoundException e) {
                e.printStackTrace();
           } catch (IOException e) {
                e.printStackTrace();
           }

           return true;
      }

}
```

Activity **AddToGallery:**

Imports:

```
import java.io.ByteArrayOutputStream;
import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.Toast;
```

Atributos:

```
final int FILE_CHOOSED = 0;
static String ip = "";
static String fileName = "";
Sharephoot.ImageServerPrx sender;
static byte[] photoReadyToSend;
```

Método **onCreate(Bundle savedInstanceState):**

```
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      initOnCreateIface();
}
```

Método **initOnCreateIface():**

```
private void initOnCreateIface(){
      setContentView(R.layout.add);

      Bundle bundle = this.getIntent().getExtras();
      ip = bundle.getString("ipIceServer");

      try {
            initIceConnection(true);
      } catch (Ice.LocalException e) {
            Toast.makeText(AddToGallery.this,
            "Your server is not active.\nPlease go to settings and set a
            new one.",
            Toast.LENGTH_SHORT).show();
            Log.e("ErrorLog","Error Local Exception");
            finish();
      }

      Button buttonChooseBack = (Button)findViewById(R.id.choose_back);
      buttonChooseBack.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
```

```
                finish();
            }
        });
        Button buttonChoose = (Button)findViewById(R.id.choose);
        buttonChoose.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {
                Intent intent = new Intent(AddToGallery.this,
        FileChooser.class);
                startActivityForResult(intent, 0);
            }
        });

}
```

Método **onActivityResult(int requestCode, int resultCode, Intent data):**

```
        // Listen for results.
protected void onActivityResult(int requestCode, int resultCode, Intent
data){

if(data!=null){
        setContentView(R.layout.add_selected);
        ImageView image = (ImageView) findViewById(R.id.photo_selected);
        BitmapFactory.Options options = new BitmapFactory.Options();

        options.inSampleSize = 6;

        Bitmap bMap =
        BitmapFactory.decodeFile(data.getStringExtra("file"),options);

        fileName = data.getStringExtra("fileName");


        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        bMap.compress(Bitmap.CompressFormat.PNG, 100, baos);
        photoReadyToSend = baos.toByteArray();
        image.setImageBitmap(bMap);

        Button buttonSend = (Button)findViewById(R.id.send);
        buttonSend.setOnClickListener(new OnClickListener(){
            public void onClick(View v) {

                initIceConnection(false);

                sender.sendName(fileName);
                sender.sendImage(photoReadyToSend);


                Toast.makeText(AddToGallery.this, "You've SharePhooted
        this photo!", Toast.LENGTH_SHORT).show();
                new Handler().postDelayed(new Runnable() { public void
        run() {
```

```
        //Re-Start this Activity for another photo
                    Bundle bundle = new Bundle();
                    bundle.putString("ipIceServer", ip);
                     Intent intent = new Intent();
                     intent.setClass(AddToGallery.this, AddToGallery.class);
                     intent.putExtras(bundle);
                    startActivity(intent);
                    finish();
                    } }, 2000);
        }
});
}else{
        initOnCreateIface();
}

}
```

Método **onCreate(Bundle savedInstanceState):**

```
private void initIceConnection(boolean timeout) throws
Ice.LocalException{
        Ice.Communicator ic = null;
        ic = Ice.Util.initialize();
        Ice.ObjectPrx base;
        if(timeout){
          base = ic.stringToProxy("ImageServerBind:tcp -h " + ip+ " -t 500
                                        -p 60000");
        }else{
          base = ic.stringToProxy("ImageServerBind:tcp -h " + ip+" -t 3000
                                        -p 60000");
        }
        sender = Sharephoot.ImageServerPrxHelper.checkedCast(base);
}
```

Anexo 7

Layouts XML de SharePhoot

Anexo 7: Layouts XML de SharePhoot

En este capítulo, se explicarán en detalle los ficheros **XML** que componen la parte gráfica de la aplicación.

Layouts Implicados en la Activity SharePhootAct

Layout **main.xml:**

Este *layout* simplemente es una imagen de bienvenida a la aplicación. Dicha imagen se encuentra en directorio de recursos "*drawable*" con el nombre de *launcher.png*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/launcher"
    >
</LinearLayout>
```

Figura 105: Resultado de main.xml

Layout **menu_launcher.xml:**

Este *layout* se compone de dos layout lineales. El primero para contener toda la vista, y el segundo para establecer los elementos. En este caso son 4 botones referentes a las 4 acciones que tiene el menú principal.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/wallpaper2">

    <LinearLayout
    android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:layout_marginTop="10dp"
    android:layout_marginRight="5dp"
    android:gravity="right"
    android:baselineAligned="true">

        <Button
        android:id="@+id/launcher_add"
        android:layout_width="90px"
        android:layout_height="90px"
        android:background="@drawable/launcher_add" />
        <Button
        android:id="@+id/launcher_gallery"
        android:layout_width="90px"
        android:layout_height="90px"
        android:background="@drawable/launcher_gallery" />
        <Button
        android:id="@+id/launcher_settings"
        android:layout_width="90px"
        android:layout_height="90px"
        android:background="@drawable/launcher_settings" />
        <Button
        android:id="@+id/launcher_exit"
        android:layout_width="90px"
        android:layout_height="90px"
        android:background="@drawable/launcher_exit" />
    </LinearLayout>
</LinearLayout>
```

Figura 106: Resultado de menu_launcher.xml

Layout **settings.xml:**

Este *layout* es un *layout* relativo. Esto quiere decir que los elementos que se mostrarán en él, deben llevar un orden y tienen que indicar si son elementos raíz o elementos seguidores. Esto se indica, por ejemplo, con el atributo *android:layout_below*. Se compone de una caja de texto y tres botones

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:paddingTop="50px"
    android:background="@drawable/background_black"
    >
    <TextView        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="IP Server Address:"/>
    <EditText
        android:id="@+id/ip_server"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"
        android:textColor="#000000"
        />
    <Button        android:id="@+id/store"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/ip_server"
        android:layout_alignParentRight="true"
        android:text="Store" />
    <Button     android:id="@+id/clear"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/store"
        android:layout_alignTop="@id/store"
        android:text="Clear" />
    <Button     android:id="@+id/back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/clear"
        android:layout_alignTop="@id/clear"
        android:text="Go back" />
</RelativeLayout>
```

Figura 107: Resultado de settings.xml

Layouts Implicados en la Activity SaveThisPhoto

Layout **save_this_photo.xml:**

Consiste en mostrar una imagen en tamaño ampliado mientras se está guardando en la tarjeta **SD** del terminal. Se muestra durante un breve periodo de tiempo junto a un mensaje informativo y luego se vuelve a la galería de imágenes.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:background="@drawable/background_black"
     >
     <TextView
         android:id="@+id/label_save"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:text="Saving your photo..."/>

      <ImageView android:id="@+id/vistaImagen"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:gravity="top"/>
</LinearLayout>
```

Figura 108: Resultado de sabe_this_photo.xml

8.C.A Layouts Implicados en la Activity AddToGallery

Layout **add.xml:**

En esta vista, hace de pantalla de presentación entre las dos *Activities* que se encargan de seleccionar la imagen (la *Activity FileChooser*) y la que muestra la imagen en grande y espera que pulsemos el botón de enviar. Consiste en un fondo diseñado específicamente para esta pantalla y dos botones, uno para volver atrás y otro para lanzar la *Activity* que elegirá el fichero.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/background_add"
    android:gravity="center|center_vertical">

    <Button
        android:id="@+id/choose"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center|center_vertical"
        android:text="Pick a pic" />
    <Button
        android:id="@+id/choose_back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center|center_vertical"
        android:text="Go menu" />
</LinearLayout>
```

Figura 109: Resultado de add.xml

Layout **add_selected.xml:**

Esta es la vista de la *Activity* de la que se hablaba en el *layout add.xml*. Ésta es la que se encarga de mostrar la imagen en tamaño aumentado y aguarda a que se pulse el botón de enviar (*send*) para conectar con el servidor *ICE* y enviarla allí. Consiste de un *layout* relativo en el cual se incluyen en orden una etiqueta con el nombre de la imagen, el botón de enviar y la imagen en cuestión.

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/background_add"
    android:paddingTop="50px"
    >
    <TextView
```

```
        android:id="@+id/add_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="\nPhooto to send"
        android:textStyle="bold"
        android:paddingRight="50px"

        />
    <Button
     android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_toRightOf="@id/add_title"
        android:text="Send it now!"
        />
     <ImageView android:id="@+id/photo_selected"
          android:layout_width="fill_parent"
          android:layout_height="300px"
          android:layout_below="@id/send"
          android:scaleType="center"
          />
</RelativeLayout>
```

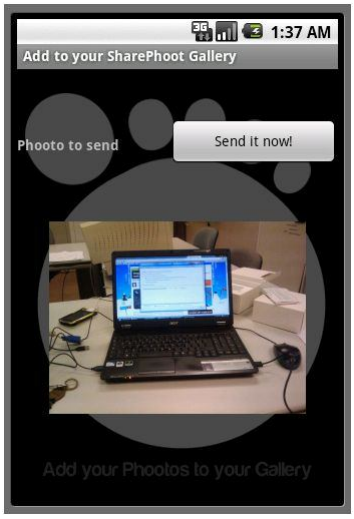


Figura 110: Resultado de add_selected.xml

Layouts Implicados en la Activity FileChooser

Layout **file_view.xml:**

Layout simple que se encarga de mostrar el nombre del fichero en primer lugar y el tamaño de este en segundo. Esta vista a pesar de su simpleza, es de gran ayuda a la hora de acceder a los contenidos visibles de la tarjeta *SD* del terminal.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_height="wrap_content"
  android:orientation="vertical"
```

```
  android:layout_width="fill_parent"
  >
      <TextView  android:text="@+id/TextView01"
            android:id="@+id/TextView01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:singleLine="true"
            android:textStyle="bold"
            android:layout_marginTop="5dip"
            android:layout_marginLeft="5dip">
      </TextView>
      <TextView  android:text="@+id/TextView02"
            android:id="@+id/TextView02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginLeft="10dip">
      </TextView>
</LinearLayout>
```

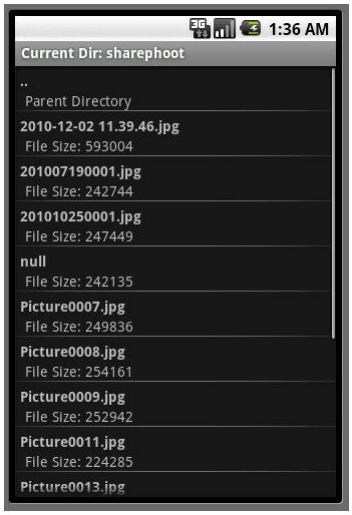


Figura 111: Resultado de file_view.xml

Layouts Implicados en la Activity Gallery

Layout **gallery_main.xml:**

Este es el layout que compone la galería de miniaturas que se tienen en un servidor remoto bajo *ICE*. Se compone de una vista de tipo "*grid*" o cuadrícula. En este caso no es necesario ningún elemento adicional para mostrar la galería como una cuadrícula ya que toda su descripción se encuentra en sus atributos.

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/gridview"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:columnWidth="100px"
      android:numColumns="auto_fit"
      android:verticalSpacing="5dp"
      android:horizontalSpacing="5dp"
```

```
      android:stretchMode="columnWidth"
android:background="@drawable/background"/>
```

Figura 112: Resultado de gallery_main.xml

Layout **view_photo.xml:**

Este layout se encarga de mostrarnos una imagen a tamaño completo tras seleccionar una miniatura de la galería. Esta vista permanece visible hasta que el usuario presiona la imagen que ha solicitado.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/background_black"
    android:paddingTop="50px"
    >
    <TextView
        android:id="@+id/label_title"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Untitled"
        android:textStyle="bold"/>
     <ImageView android:id="@+id/photo_full"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_below="@id/label_title"
          android:scaleType="fitStart"
/></RelativeLayout>
```

Figura 113: Resultado de view_photo.xml

Layout **menu.xml:**

Menú que se muestra cuando en la galería, el usuario presiona o selecciona una imagen o en su defecto, pulsa la tecla "*Menu*" de su terminal.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
          android:color="#677D00">
    <item android:id="@+id/ic_save"
          android:icon="@drawable/ic_save"
          android:title="@string/ic_save" />
    <item android:id="@+id/ic_view"
          android:icon="@drawable/ic_view"
          android:title="@string/ic_view" />
    <item android:id="@+id/ic_refresh"
          android:icon="@drawable/ic_refresh"
          android:title="@string/ic_refresh" />
    <item android:id="@+id/ic_delete"
          android:icon="@drawable/ic_delete"
          android:title="@string/ic_delete" />
     <item android:id="@+id/ic_cancel"
          android:icon="@drawable/ic_cancel"
          android:title="@string/ic_cancel" />
</menu>
```

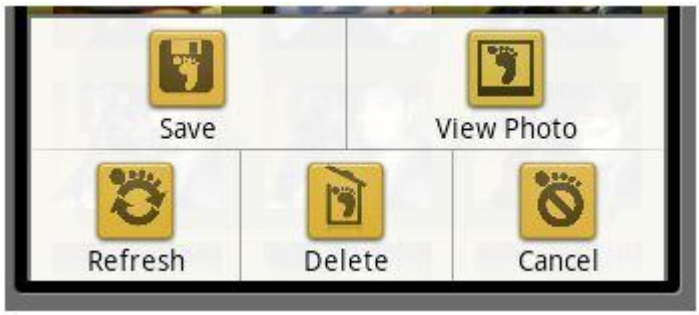


Figura 114: Resultado de menu.xml

Anexo 8

Android Manifest de SharePhoot

Anexo 8: Android Manifest de SharePhoot

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="it.uc3m.SharephootClient"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name=".SharePhootAct"
                  android:label="@string/app_name"
android:screenOrientation="portrait"
                  android:configChanges="orientation|keyboardHidden">
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
                 <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".Gallery"
                      android:label="@string/myGallery"
android:screenOrientation="portrait"

android:configChanges="orientation|keyboardHidden">
                      <intent-filter>
                 <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".AddToGallery"
                      android:label="@string/addGallery"
android:screenOrientation="portrait"

android:configChanges="orientation|keyboardHidden">
                      <intent-filter>
                 <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".SaveThisPhoto"
                      android:label="@string/savePhoto"
android:screenOrientation="portrait"

android:configChanges="orientation|keyboardHidden">
                      <intent-filter>
                 <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".FileChooser"
                      android:label="@string/fileChooserTitle"
android:screenOrientation="portrait"

android:configChanges="orientation|keyboardHidden">
                      <intent-filter>
```

```
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```